Proactive and reactive runtime service discovery: a framework and its evaluation

Proactive and Reactive Runtime Service Discovery: A Framework and its Evaluation

A. Zisman, G. Spanoudakis, J. Dooley, I. Siveroni

Abstract — The identification of services during the execution of service-based applications to replace services in them that are no longer available and/or fail to satisfy certain requirements is an important issue. In this paper we present a framework to support runtime service discovery. This framework can execute service discovery queries in pull and push mode. In pull mode, it executes queries when a need for finding a replacement service arises. In push mode, queries are subscribed to the framework to be executed proactively, and in parallel with the operation of the application, in order to identify adequate services that could be used if the need for replacing a service arises. Hence, the proactive (push) mode of query execution makes it more likely to avoid interruptions in the operation of service-based applications when a service in them needs to be replaced at runtime. In both modes of query execution, the identification of services relies on distance-based matching of structural, behavioural, quality, and contextual characteristics of services and applications. A prototype implementation of the framework has been developed and an evaluation was carried out to assess the performance of the framework. This evaluation has shown positive results, which are discussed in the paper.

Index Terms — Web-services discovery, Composite web services, Context-Aware QoS Model, Application development in services.

1 Introduction

Service-based applications are composed of loosely coupled autonomous computer-based entities owned by third parties known as services. These services are combined to realize applications and create dynamic business processes. Due to rapid changes in market conditions and regulations, the dynamic creation of business alliances and partnerships, and the need to assist with changing user demands, it is necessary to provide ways of identifying services that can fulfill specific functional and quality characteristics of service-based applications. The identification of such services is known in the literature as *service discovery* and has been an important topic of research over the last few years.

Several approaches have been developed to support service discovery, broadly classified as *static* [20][24][30][47] and *dynamic* [10][13][38][48] approaches. In static service discovery, services are identified during the development of service-based applications and bound to these applications prior to execution. In dynamic (aka runtime) service discovery, services are identified and bound to service-based applications during the execution of the applications. This may be necessary in order to replace existing services in an application and allow the application to continue its execution.

The need to replace services during the execution of a service-based application may arise due to different circumstances such as (a) the unavailability or malfunctioning of a service used in the application; (b) changes in the structure (i.e., interface), functionality, quality properties, or the context of services used in the application that make them no longer appropriate or the best option for the role they fulfill; (c) changes in the context of an application that can also make used services no longer appropriate or the best option for the role they fulfill; or (d) the emergence of new services that can fulfill the role of an existing service in an application in a better way than the current service. In the above cases, the term "context" signifies information about the operational environment of an application or a service that changes dynamically (e.g., location, workloads, network availability) and can affect the adequacy of a service for an application.

The above circumstances give rise to a basic research challenge, i.e., how to support service-based applications when the services that they use disappear or stop functioning as expected, as well as in the presence of continuously changing contexts of both the applications and their services at runtime. Addressing this challenge requires a dynamic and flexible identification of services during the execution time of service-based applications.

Most of the current approaches for dynamic service discovery support a classic pull mode of query execution. This mode is often not effective. This is because the discovery process is triggered only after the need for a new service arises (as in case (a) above) and it may take considerable time to complete, affecting the performance of the application and its ability to produce acceptable "real time" response to the user. It should also be noted that

- *A. Zisman is with School of Informatics, City University London, Northampton Square, London EC1V 0HB, UK, E-mail: a.zisman@soi.city.ac.uk.*
- *G. Spanoudakis is with School of Informatics, City University London, Northampton Square, London, EC1V 0HB, UK, E-mail: G.E.Spanoudakis@city.ac.uk.*
- *J. Dooley is with School of Computer Science and Electronic Engineering, University of Essex, Colchester CO4 3SQ, UK, E-mail: jpdool@essex.ac.uk.*
- *I. Siveroni is with School of Informatics, City University London, Northampton Square, London EC1V 0HB, UK, E-mail: sbbc287@soi.city.ac.uk.*

Manuscript received (insert date of submission if desired). Please note that all acknowledgments should be placed at the end of the paper, before the bibliography.

pull mode discovery cannot identify better services (as in case (d) above) before a problem with an existing service that would trigger the execution of a query arises. Similarly, for cases (b) and (c) above, pull mode discovery would need to wait until the changes that made used services be inadequate arise at runtime as in case (a). Alternatively, the pull mode of query execution would need to be enhanced with mechanisms for polling regularly service registries and/or context information resources to identify changes that can lead to subsequent problems. Such polling would consume significant computational resources, as it would need to be executed at regular intervals even if there is no need to do so (i.e., in the absence of service context changes, application environment context changes, or emergence of new services).

Furthermore, existing approaches to service discovery (with the exception of [48]) do not consider different characteristics of the application such as structural, behavioural, quality, and contextual aspects, at the same time when attempting to identify services.

To address the limitations of existing approaches, we present a service discovery framework that supports runtime service discovery based on complex queries that can express flexible combinations of structural, behavioural, quality, and contextual conditions. These queries are specified in an XML-based query language, called *SerDiQueL*. The framework assumes services that have multi-faceted descriptions including service interface, behaviour, quality, and context descriptions.

To support all cases (a) to (d) above and avoid the drawbacks of traditional polling mechanisms, our framework allows service discovery based on both reactive (pull) and proactive (push) query execution modes. Pull mode query execution is triggered in cases like (a) above. In push mode, query execution is performed in parallel to the execution of the application using pre-subscribed queries. These queries are associated with specific services in an application and aim to maintain up-to-date sets of candidate replacement services for these services. In both modes, query execution is based on matching and the computation of distances between query and service specifications.

The work presented in this paper has been carried out as part of a European research project focusing on the development of service based grid applications (GREDIA [19]) and is based on scenarios identified in different industrial domains including media and banking. Previous work on our runtime discovery framework has been presented in [14][32][49][61][62]. The new version of the runtime framework presented in this paper incorporates two main extensions. The first of these extensions is the development of an XML-based query language to support the specification of service discovery queries of both push and pull types, including the specification of service behavioural conditions in queries. The language for expressing behavioural query conditions was introduced to replace the specification of such conditions using BPEL in earlier versions of the framework [62]. The new language allows for the declarative specification of partial behavioural conditions for services rather than requiring the specification of complete procedural models of expected service behaviour, as in the original version of the framework, that that was cumbersome.

The second extension of the framework is the development of a new behavioural matching process and the use of a new behavioural distance for evaluating the behavioural conditions expressed in the new query language. Furthermore, in this paper we give an integrated description of the framework covering all its parts and present the results of a thorough performance evaluation of it. We also provide a critical comparison of the framework in the context of related literature.

The remainder of this paper is structured as follows. Section 2 presents descriptions of discovery scenarios that will be used throughout the paper to illustrate the work and an overview of how our discovery framework supports these scenarios. Section 3 describes the service discovery query language of the framework. Section 4 presents the service discovery process supported by the framework. Section 5 presents an evaluation of the approach. Section 6 discusses related work and, finally, Section 7 presents conclusions and plans for of future work.

2 Overview

In this section, we present scenarios for runtime service discovery and give an overall description of how our framework can be used to address these scenarios.

2.1 Runtime service discovery scenarios

Various scenarios regarding runtime service discovery can be identified in reference to a mobile service-based application, called *on-the-go-News*.

on-the-go-News allows its users to request and receive news from different media sites from their mobile phone. To do so, the application offers services allowing users to: (i) search for certain news topics on a mobile phone and choose the source which they want to receive the news from; (ii) display news about a topic from various sources; (iii) create customized on-the-fly "magazines" or with information from several different news sites; (iv) flip through articles in a customized magazine from several sources; (v) obtain and pay for the non freely available information charging the amount in the user's phone bill at the end of the month, and (vi) see the new balance of their phone bill after using the application for (v).

on-the-go-News uses an external service, called S_{Search}, which searches different news sites to identify news about specific topics, and another service, called $S_{CustMag}$, enabling the amalgamation of news and their customized appearance in an on-the-fly magazine.

One runtime service discovery scenario can arise if after receiving a request for news on a specific topic, *on-the-go-News* fails to contact S_{Search} due to the fact that the latter service is unavailable (Case (a)). In this case, the application will need to identify a new service to replace S_{Search}.

After the new service is identified and bound to *on-the-go-News*, the user who issued the request will start receiving the requested information from various sites.

A second scenario, may arise if a user who is interested on having an on-the-fly magazine about climate change on his mobile phone and created such a magazine using $S_{CustMag}$, starts getting a slow response from $S_{CustMag}$ as the service is used by many different users simultaneously (Case (b)). In such cases, an alternative service for $S_{CustMag}$ with acceptable response time will need to be identified and bound to *on-the-go-News*.

A third scenario arises when, whilst a user of *on-the-go-News* is travelling by train, he looses access to the service that displays and supports flipping through news, (i.e., a service called $S_{DisFlip}$) since $S_{DisFlip}$ cannot be accessed at his current location. This change in the location of *on-the-go-News* (Case c) requires searching for an alternative service that could be used in the user's current location.

A fourth scenario arises when a new service that allows payments by debiting the user's bank account and credit card payments, instead of charging the user's phone bill becomes available (Case d). If flexibility in payment is desirable in *on-the-go-News*, the new service should be bound to the application.

2.2 Framework support

To support cases (a)-(d) above, *on-the-go-News* will need to perform different types of runtime service discovery. In case (a), there will be a need to discover a new service following an exception at the point where the application tries to call S_{Search}. In case (b), it will need to discover a new service to replace $S_{CustMag}$ when a deterioration of the performance of this service is detected. In case (c), it will need to identify services that can only be used when the device on which the application runs is at specific location(s). Finally, in case (d) it will need to identify that a new service that can be used in it has emerged and should be used, as it has a better fit with the service discovery criteria, to enhance the overall level of service to the user. The former three of these discovery scenarios could be undertaken re-actively (i.e., after the event/problem that signals the discovery need occurs) or proactively in order to ensure minimal interruption of service when the problem occurs. The discovery action related to the fourth case (i.e., case (d)) needs to be taken proactively as the emergence of a new service is not associated with any problem in the operation of the service based application.

Our framework enables service-based applications, like *on-the-go-News* to address such runtime discovery scenarios both in a reactive and a proactive manner without having to incorporate code implementing the discovery functionality required in each case. To achieve this, an application must:

(i) register to the framework a list of service endpoints that it wishes to use (and potentially replace) along with one query for each such service that should be used for discovering alternatives to it; and

(ii) call operations of the registered services through the framework.

Actions (i) and (ii) are realized by an API that is available through the discovery framework. Action (i) must take place at the start of each execution of the application but will have no new effect, if it has already been executed previously.

At runtime when the application calls an operation of a registered service, the framework accepts the call and tries to call the relevant operation. If the service that provides the operation is not available and the second call fails, the framework will attempt to respond to the application request by calling a corresponding operation of an alternative service. The operation to be called is determined by the execution of the query. More specifically, if the query has been subscribed as a reactive (pull) mode query, the framework will execute the query immediately after the failed call to S and if it can find an alternative service with a suitable operation it will call this operation. If the query has been subscribed as a proactive (push) mode query, the framework will have proactively executed the query and built a set of possible alternative services for S by the time of the failed call[1]. Thus, if S is unavailable and the call fails, the framework will select the best service in the already built alternative set of services (say service S'), make a call to a corresponding operation of S', and respond back to the application when it receives a response from S'. Following this initial replacement, S' will continue to be used in the place of S until an event that makes it necessary to replace S' occurs.

The query associated with a service S will be used to identify alternative candidate services for S that could be used anytime that a need to replace S within an application arises, regardless of the exact event that signaled the need for the replacement of S (e.g., failed call to an operation of S, deterioration of the performance of S, unavailability of S due to location changes).

Figure 1 shows the overall architecture of the runtime service discovery framework, shortly referred to as RSDF in the following, and its main components. These components are an *execution engine*, a *service requester*, a *service matchmaker*, a *service listener*, *service* and *application context servers*, and a *service registry intermediary*.

The *service requester*① orchestrates the functionality offered by the other components in the framework. It receives a service request from a client *service-based application*② as well as context information about the services and application environment, (b) prepares service queries to be evaluated, (c) organises the results of a query into an ordered set of matching services, (d) manages push query execution mode subscriptions, (e) receives information from listeners about new services that become available or changes to existing services, and (f) invokes the other components to execute a query.

[1] If a call is made so early that no proactive execution of the query has taken place yet, the query will be executed reactively for the first time and proactively from that point onwards.

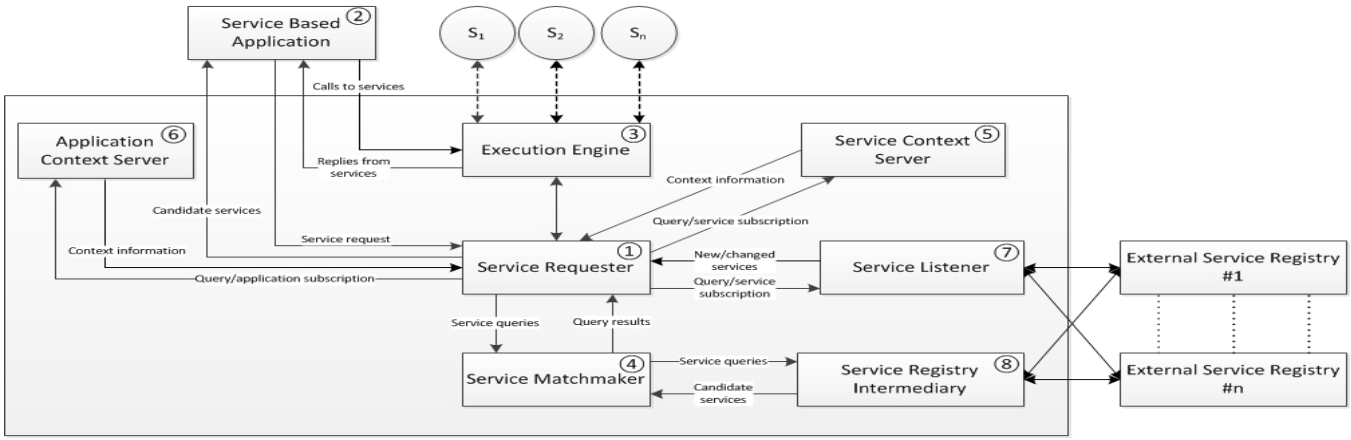

Fig 1: Framework Architecture

Service-based applications make a call to a service through the *execution engine*③. After receiving a request for a service, the execution engine retrieves the actual service endpoint from the service requester and calls the service. When the service replies, the execution engine forwards the reply to the application.

The *service matchmaker*④ parses service discovery queries and evaluates them against service specifications in the various service registries (see Sec. 4.1).

The *service* and *application context servers* (⑤ and ⑥ respectively) support the acquisition of context information about the services and the application environment, respectively. Both context servers accept subscriptions for specific types of context information from the service requester and send updates when changes in the context of services and application occur. These two servers can be deployed in different machines from the ones where the application and services are deployed and even implemented as services.

The *service listener*⑦ sends to the service requester notifications about new services that become available, or about changes in the descriptions of existing services. This information is extracted from external service registries through polling. The notifications are based on subscriptions for specific types of information that the service requester has made to the service listener. Following the

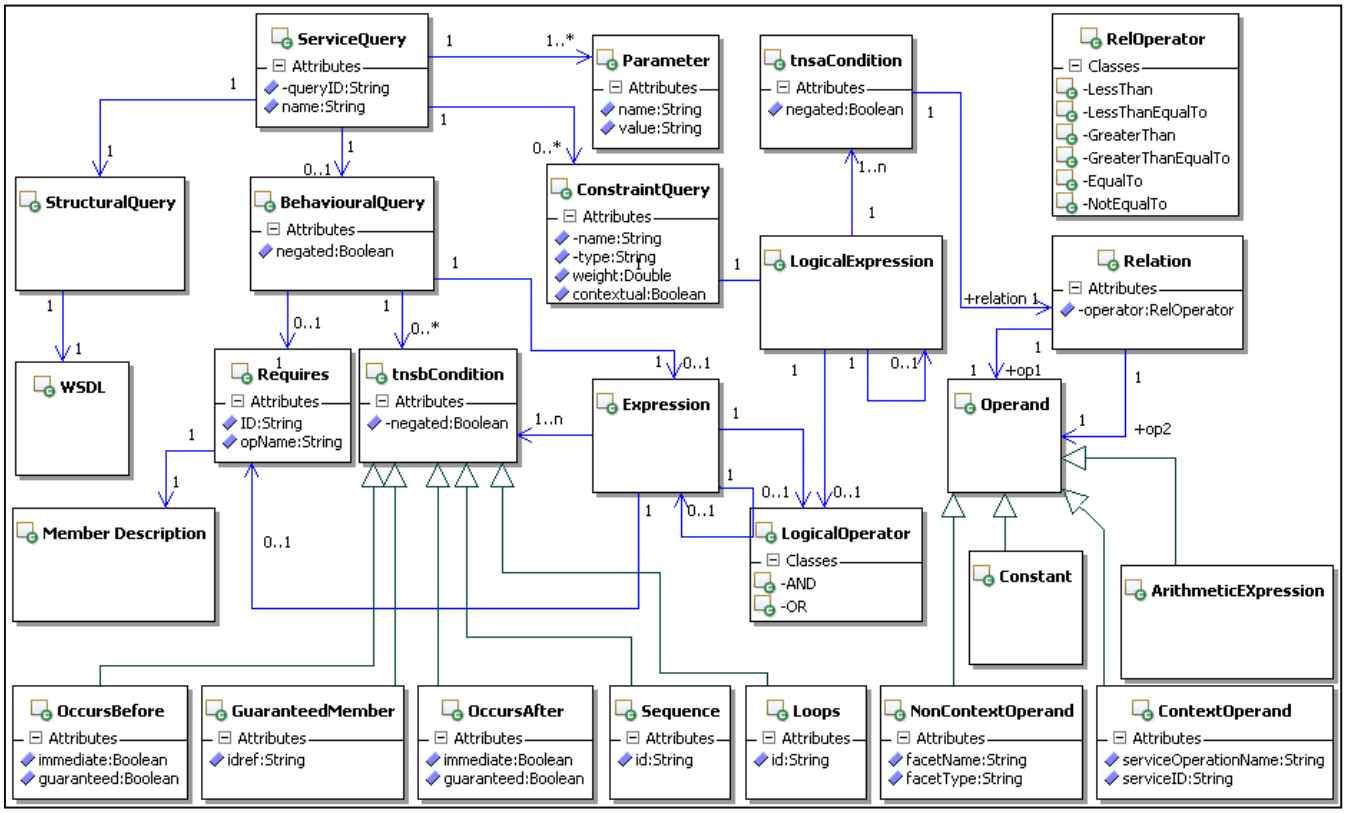

Fig 2: UML representation of SerDiQueL

notification of a new service, the service requester evaluates whether the new service matches with any of the queries and, if it does, the requester notifies the service context server so that the context conditions of the service can subsequently be observed.

Finally, RSDF incorporates a *service registry intermediary*® supporting the use of different service registries and the discovery of services stored in them by providing an interface for accessing the registries. The current implementation of RSDF supports registries that are based on the faceted service description scheme developed in the SeCSE project [44]. In this scheme, a service is specified by a set of XML facets representing different service aspects, including (i) structural facets describing the operations of services with their data types using WSDL [55], (ii) behavioural facets describing behavioural models of services in BPEL [8], (iii) quality of service facets describing quality aspects of services represented in XML-based schemas, and (iv) context facets describing the types of context information that are available for a service and operations.

3 Service Discovery Query Language

A runtime service discovery query may contain different criteria, namely: (i) *structural criteria*, describing the interface of the required service; (ii) *behavioural criteria*, describing the functionality of the required service; and (iii) *constraints*, specifying additional conditions for the service to be discovered. The latter conditions may refer to quality aspects of the required service or interface characteristics of services that cannot be represented by the standardised forms of structural descriptions used in the framework. Examples of constraints referring to quality characteristics of services may concern the maximum response time or cost to execute a certain operation in a service. Sec. 3.3 provides examples of additional structural constraints.

The constraints in a query can be *contextual* or *non-contextual*. A contextual constraint is concerned with information that changes dynamically during the operation of the service-based application or the services that the application deploys, while a non-contextual constraint is concerned with static information. The constraints can be hard or soft. A hard constraint must be satisfied by all discovered services for a query and is used to filter services that do not comply with them. Soft constraints do not need to be satisfied by all discovered services, but are used to rank candidate services.

To specify runtime service discovery queries, we have developed an XML-based language, called *SerDiQueL*. *SerDiQueL* allows the specification of all the structural, behavioural, quality and contextual characteristics required from the services to be discovered. An earlier version of *SerDiQueL* was presented in [49]. The new version of the language that we present in this paper enables the specification of the required behaviour of a service using behavioural conditions rather than a full BPEL model of service behaviour as in the original version.

Figure 2 gives an overall representation of *SerDiQueL* as a UML class diagram. As shown in the figure, a *SerDiQueL* query (*ServiceQuery*) has a unique identifier (*queryID*) and a *name*, and is composed of one or more elements describing different *parameters* for a query and three other elements representing the *structural*, *behavioural*, and *constraint* sub-queries.

A parameter element is defined by a name and a value. Examples of parameters that can be currently used in a query are the query (a) name, (b) type (dynamic or static), (c) mode of execution (push or pull), (d) author, and (e) distance threshold for selecting the set of candidate replacement services (see Sec. 4). Below, we describe the main constructs for specifying structural, behavioural, and constraint sub-queries in *SerDiQueL*. However, a detailed description of the XML schema of the language is beyond the scope of this paper and can be found in [16].

In order to illustrate the use of *SerDiQueL*, let us reconsider *on-the-go-News*, the service-based application described in Sec. 2.1, and a service $S_{Payment}$ that is used by *on-the-go-News* to take payments for received news by transferring money from the user's bank account, after checking for the account's balance ($S_{Payment}$ is similar to *PayPal* [43]). If $S_{Payment}$ becomes unavailable, a query would need to be executed to find an adequate replacement service for $S_{Payment}$. Let us assume that this query would need to express the following discovery criteria:

(i) The service should authenticate its user before allowing access to its functionality.
(ii) The service should be provided by *"Banca Populare di Sondrio"* (POPSO).
(iii) The service should be available 24 hours a day, and all the necessary actions for taking a payment should take no more than 5 seconds to be executed.

3.1 Structural Sub-query

The structural sub-query describes the interface of the required service. Structural sub-queries in *SerDiQueL* are specified using WSDL [55]. The use of WSDL in this case is due to its wide acceptance as a service interface description language. In addition, during runtime service discovery, any replacement service that might be identified for an existing service in a service-based application will need to conform to the interface of the existing service. A structural sub-query in *SerDiQueL* is specified as the WSDL specification of the service to be replaced.

Figure 3 shows an extract of a *SerDiQueL* query (i.e., query Q1) for identifying services that could replace service $S_{Payment}$ in *"on-the-go-News"*. As specified by the *Parameter* elements *type, mode* and *threshold* in the query, Q1 is a dynamic type query (i.e., a query that is executed at runtime) of push mode, with a distance threshold of 0.8. For simplicity and due to space limitations, Figure 3 does not present the full structural sub-query that should be used for finding a replacement for $S_{Payment}$ and includes only a part of it indicating the portType (i.e., the interface)

required of adequate replacement services, the messages used to interact with the operations in this interface, and the structure of the data included in these messages. It should be noted, however, that the structural subquery in this case is the WSDL description of $S_{Payment}$.

3.2 Behavioural Sub-query

Behavioural sub-queries in *SerDiQueL* support the specification of (a) the existence of required functionalities in a service specification; (b) the order in which the required functionalities should be executed by the service; (c) dependencies between functionalities (e.g. a functionality realized by an operation always requires the existence of a functionality of another operation); (d) preconditions; and (e) loops concerning execution of certain functionalities. Behavioural sub-queries are expressed by elements that are similar to temporal logic operators.

```
<?xml version="1.0" encoding="utf-8"?>
<tns:ServiceQuery …queryID="Q1" name="FindBankTransferService">
<tns:Parameter name="mode" value="PUSH" />
<tns:Parameter name="type" value="dynamic" />
<tns:Parameter name="threshold" value="0.8" />
<tns:StructuralQuery>
<definitions xmlns:tns="http://samples.otn.com"
      xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/" …
      xmlns:format="http://schemas.xmlsoap.org/wsdl/formatbinding/" target
      Namespace="http://samples.otn.com">
   <types>
    <schema xmlns="http://www.w3.org/2001/XMLSchema" at
     tributeFormDefault="qualified" elementFormDefault="qualified" target
     Namespace="http://samples.otn.com">
    <complexType name="NoAccountIdExceptionType">
     <sequence> <any /> </sequence> </complexType>
    <complexType name="NotEnoughBalanceExceptionType">
     <sequence> <any /> </sequence> </complexType>
   </schema> </types>
  <!-- message declns -->
    <message name="LoginRequestMessage">
      <part name="userID" type="xsd:string" />
      <part name="password" type="xsd:string" /> </message>
    <message name="LoginResponseMessage">
      <part name="success" type="xsd:boolean" /> </message>
    <message name="TransferAmountRequestMessage">
      <part name="fromAccountID" type="xsd:string" />
      <part name="toAccountID" type="xsd:string" />
      <part name="amount" type="xsd:double" /> </message>
   <message name="TransferAmountResponseMessage">
      <part name="success" type="xsd:boolean" /> </message> …
  <!-- port type declns -->
    <portType name="PaymentService">
      <operation name="login">
      <input message="tns:LoginRequestMessage" name="LoginRequest" />
      <output message="tns:LoginResponseMessage"
              name="LoginResponse" /> </operation>
      <operation name="transferAmount">
        <input message="tns:TransferAmountRequestMessage"
              name="TransferAmountRequest" />
       <output message="tns:TransferAmountResponseMessage"
              name="TransferAmountResponse" /> </operation>
    </portType>
    </definitions>
</tns:StructuralQuery> . . .
```

Fig 3: Example of general and structural parts of SerDiQueL queries

As shown in Figure 2, a behavioural sub-query in *SerDiQueL* can be composed of: (a) a *requires* element; (b) a single condition, a negated condition, or a conjunction of conditions; or (c) a sequence of expressions separated by logical operators.

Requires elements are used to describe the service operations that need to exist in service specifications. Every query must describe one or more required service operations, represented by *MemberDescription* elements in the query (*MemberDescription* elements can be used in various conditions and expressions in a query). A member element has three attributes, namely (a) *ID*, indicating a unique identifier for the member within a query; (b) *opName*, specifying the name of an operation described in the structural sub-query; and (c) *synchronous*, indicating if the service operation needs to be executed in a synchronous or asynchronous mode in the service.

The existence of *requires* elements in service specifications is verified as an initial step during the execution of a behavioural sub-query rather than during the evaluation of the conditions and expressions of the query that use these elements. This optimizes the query execution process as there is no need to evaluate any condition or expression of a behavioural sub-query that refers to a non-existent *requires* element.

Figure 4 shows the behavioural sub-query for query Q1. As shown in the figure, Q1 includes *Requires* elements expressing the requirement for the existence of the following operations, in any replacement service, specified by *MemberDescription* elements:

- *login(userID:string, password:string):boolean*
- *credit(accountId:string, amount:double):balance*
- *transferAmount(fromAccountID:string, toAccoutID:string, amount:double):boolean*
- *debit(accountId:string, amount:double):balance*
- *getBalance(accountId:string):balance*
- *logout((userID:string):Boolean*

As shown in Figure 2, a condition is defined as a *GuaranteedMember*, *OccursBefore*, *OccursAfter*, *Sequence*, or *Loop* element. A *GuaranteedMember* represents a member element (i.e., a service operation) that needs to occur in all possible traces of execution in a service. This element is defined by the attribute *IDREF* that can reference *requires*, *sequence*, or *loop* elements. *OccursBefore* and *OccursAfter* elements represent the order of occurrence of two member elements (i.e., *Member1* and *Member2*). Note that in some cases we may require *OccursBefore(m1,m2)* whilst in other cases we may require *OccursAfter(m1,m2)*, or even need to differentiate an *OccurBefore* condition by attributes such as *immediate*. Hence both the *OccursBefore* and *OccursAfter* elements are needed. Furthermore, they have two boolean attributes, namely the attributes *immediate* and *guaranteed*. The first of these attributes specifies if two operations need to occur in direct sequence or if there can be other operations in between them. The second attribute specifies if the two operations need to occur in all possible execution traces of a service.

A *Sequence* element defines two or more members that must occur in a service in the order represented in the sequence. It has an identifier attribute that can be used by

the *GuaranteedMember*, *OccursBefore*, *OccursAfter*, *Sequence*, and *Loop* elements. A *Loop* element specifies a sequence of operations that is executed several times. It has a unique identifier (attribute *ID*) and is defined as a statement that references other identified elements (see element *Body*).

```
<tnsb:BehaviourQuery> <tnsb:Requires>
  <tnsb:MemberDescription ID="login" opName="login"  … />
  <tnsb:MemberDescription ID="credit" opName="credit"  … />
  <tnsb:MemberDescription ID="xfer" opName="transferAmount" …  />
  <tnsb:MemberDescription ID="debit" opName="debit"  … />
  <tnsb:MemberDescription ID="balance" opName="getBalance" …/>
  <tnsb:MemberDescription ID="logout" opName="logout"  …/>
 </tnsb:Requires>
 <tnsb:Expression>
  <tnsb:Condition> <tnsb:GuaranteedMember IDREF="login" />
  </tnsb:Condition></tnsb:Expression>
 <tnsb:LogicalOperator operator="AND" />
 <tnsb:Expression> <tnsb:Condition> <tnsb:Sequence ID="pay">
  <tnsb:Member IDREF="credit" /> <tnsb:Member IDREF="xfer" />
  <tnsb:Member IDREF="debit" /> <tnsb:Member IDREF="balance" />
  </tnsb:Sequence> </tnsb:Condition>
 <tnsb:Condition>
  <tnsb:OccursBefore immediate="false" guaranteed="false">
   <tnsb:Member1 IDREF="login" /><tnsb:Member2 IDREF="pay" />
  </tnsb:OccursBefore> …</tnsb:BehaviourQuery>
```

Fig 4: Example of behavioural part SerDiQueL queries

Figure 4 shows examples of some condition types. In particular,

(a) The operation *login* is defined as a *GuaranteedMember* element given that the user of the bank service needs to be authenticated (i.e., *login* operation needs to occur in all possible paths of execution in the service).
(b) The operations *credit, transferAmount, debit,* and *balance* need to be executed in this order and, therefore, they are defined in a *Sequence* element.
(c) The operation *login* should be executed before the sequence of operations in (b) specified in element *OccursBefore*.

In behavioural sub-queries, expressions are defined as sequences of *requires* elements, conjunctions or disjunctions of conditions, or nested expressions connected by logical operators AND and OR (cf. Figure 2). The definition of *requires* elements within an expression (E1) enables the specification of queries in which the non-existence of *requires* elements in a service should not invalidate its selection, if other expressions in the sub-query that are disjointed with expression E1 (i.e., expressions connected to E1 by logical operator OR) are satisfied by the service.

As we discussed previously, originally *SerDiQueL* supported behavioural queries expressed in BPEL. However, expressing the behaviour needed by a service in BPEL turned out to be difficult, as it required the specification of complete behavioural models of services, as opposed to, partial behavioural conditions that services need to satisfy. It is, for instance, easier to specify conditions requiring that a replacement for $S_{Payment}$ service should have a "credit" and a "debit" operation (*Condition-1*) and that "credit" must always have been executed prior to any execution of "debit" (*Condition-2*) rather than specifying a full behavioural model of $S_{Payment}$ to express the same conditions. Also, in some instances, the specification of a full behavioural model might not be able to express the intended meaning of the required behavioural conditions. In our example of $S_{Payment}$, for instance, specifying a BPEL process in which there is a sequence of an invocation of the "credit" operation followed by the invocation of a "debit" operation in order to express *Condition-2* would not be adequate as the query would disregard services satisfying the condition, if these services had behavioural models in which other operations could also be executed between the operations "credit" and "debit". Due to these reasons, BPEL is no longer used in the expression of behavioural subqueries in RSDF.

3.3 Constraint Sub-query

As shown in Figure 2, a constraint sub-query in *SerDiQueL* is defined as a single logical expression, or a conjunction/disjunction of two or more logical expressions, combined by logical operators AND and OR, or a negated logical expression.

```
<tnsa:ConstraintQuery name="C1" type="HARD" contextual="false" …>
 <tnsa:LogicalExpression>
  <tnsa:Condition relation="EQUAL-TO"> <tnsa:Operand1>
   <tnsa:NonContextOperand facetName="Qos" facetType="QoS">
    //QoSCharacteristic[Name="Organisation"]/Constant  …
   <tnsa:Operand2><tnsa:Constant type="STRING">POPSO
    </tnsa:Constant></tnsa:Operand2></tnsa:Condition>…
</tnsa:ConstraintQuery>
<tnsa:ConstraintQuery name="C2" type="SOFT"  contextual="false" …>
 <tnsa:LogicalExpression>
  <tnsa:Condition relation="EQUAL-TO"> <tnsa:Operand1>
   <tnsa:NonContextOperand facetName="QoS" facetType="QoS">
    //QoSCharacteristic[Name="Availability"]/Metrics/
     Metric[Name="OpenTime"][Unit="Hours"]/MinValue
   </tnsa:NonContextOperand> </tnsa:Operand1>
   <tnsa:Operand2><tnsa:Constant type="NUMERIC">00:00
    </tnsa:Constant></tnsa:Operand2> </tnsa:Condition>
  <tnsa:LogicalOperator>AND</tnsa:LogicalOperator>
  <tnsa:LogicalExpression> <tnsa:Condition relation="EQUAL-TO">
 <tnsa:Operand1><tnsa:NonContextOperand facetName="QOS"
    facetype="QOS">//QoSCharacteristic[Name="Availability"]/Metrics/
    Metric[Name="OpenTime"][Unit="Hours"]/MaxValue
  </tnsa:NonContextOperand> </tnsa:Operand1> <tnsa:Operand2>
  <tnsa:Constant type="NUMERIC">24:00</tnsa:Constant>
  </tnsa:Operand2> </tnsa:Condition> </tnsa:LogicalExpression>
 </tnsa:LogicalExpression></tnsa:ConstraintQuery>
```

Fig 5: Example of non-contextual constraints for query Q1

Query constraints have four attributes: (a) *name*, specifying the name of the constraint; (b) *type*, indicating whether the constraint is hard or soft; (c) *weight*, specifying the significance of the constraint as a real number in the range [0.0, 1.0]; and (d) *contextual*, indicating whether the constraint refers to a contextual or non-contextual feature of a service. The weight of a constraint is used to prioritise it against other soft constraints when inexact matches are found in query evaluation. A constraint sub-query whose *contextual* attribute is *true* contains *ContextOperand* elements. When this attribute is set to *false*, the query may only contain *NonContextOperand* elements.

A logical expression is defined as a condition, or logical combination of conditions, over elements or attributes of service specifications or context aspects of operations.

Conditions are defined as relational operations expressing comparisons between the values of two operands (*operand1* and *operand2*). The supported comparisons are specified by elements expressing the relational operations *equalTo*, *notEqualTo*, *lessThan*, *greaterThan*, *lessThanEqualTo*, *greaterThanEqualTo* and *notEqualTo* (see [16]). These operations have the normal respective meanings for comparisons between their operands. The operands can be non-contextual operands, contextual operands, arithmetic expressions, or constants.

A *non-contextual operand* (i.e., an element of type NonContextOperand) has information about the name and the type of the service specification facet from which the operand's value will be retrieved during the constraint evaluation, and an XPath expression indicating elements and attributes in the service specification facet. The evaluation of this XPath expression will provide the value of the non-contextual operand during the evaluation of the enclosing constraint. Hence, constraints can be specified against any element or attribute of any facet in the description of a service in a registry.

Figure 5 shows examples of non-contextual constraint sub-queries for query Q1 above. The constraint sub-query C1 in the figure, for example, is a hard non-contextual constraint expressing that the provider of the new service should be POPSO, as described in the element *Organisation* of facet QoS which describes information about the provider of the service.

The second constraint sub-query (C2) in Figure 5 is a soft non-contextual constraint representing the fact that the service to be identified needs to be available 24 hours a day. As shown in Figure 5, this constraint has a weight of 0.5 and is represented by the conditions that verify if the opening time hours specified in the facet QoS has a minimum value of 00:00 and a maximum value of 24:00. This is specified by a conjunction of two LogicalExpression elements with their respective XPath expression contents and constant sub-elements.

As shown in Figure 2, a *contextual operand* (i.e., an element of the type *ContextOperand*) indicates the operations that can be invoked to provide context information at runtime (aka context operations). A contextual operand describes the semantic category of a context operation instead of its exact signature. This category is represented by the sub-element *ContextCategory*. The reference to operation categories rather than exact signatures is due to the fact that context operations may have different signatures across different services even if they exist to provide context information of the same type. Thus, when evaluating context conditions it should be possible to invoke such operations, despite their different signatures, in order to obtain the same type of context information for different services and evaluate the contextual constraints.

A *ContextCategory* element represents the semantic category of an operation, instead of its actual signature. A *ContextCategory* is defined as a condition about the description of the category of the operation. This description is included in a *context facet* associated with the operation. The context facet makes reference to an ontology document. To express the condition, the *ContextCategory* in a query contains an XPath expression referencing an element in the ontology document. The condition is a relational condition between the value of this element and a constant. The language can support different ontologies for describing context operation categories since it does not make any assumption of the structure and meaning of the ontologies used, apart from the fact that the ontologies need to be described in XML. The evaluation of the query verifies if a candidate service has a context operation with a semantic category that satisfies the condition.

A contextual operand is further defined by: (a) an attribute specifying the name of the service operation associated with the operand (*serviceOperationName*), and (b) an attribute specifying the identifier of a service that provides this operation (*serviceID*). The value of the latter attribute is specified when the context operand provides the specification of a context operation of a known service. This is normally the case when the context operation is associated with a service-based application for which the value of a context aspect of the application needs to be dynamically identified during the evaluation of a query (e.g., location of a mobile device application). In this case, attribute *serviceID* refers to the service-based application. Otherwise, the value of *serviceID* is "any".

```
<tnsa:ConstraintQuery name="C3"  contextual="true" type="SOFT" …>
 <tnsa:LogicalExpression>
 <tnsa:Condition relation="LESS-THAN-EQUAL-TO"><tnsa:Operand1>
   <tnsa:ContextOperand serviceOperatioName="transferAmount">
     <tnsa:ContextCategory relation="EQUAL-TO"><tnsa:Category1>
  <tnsa:Document location="http://eg.org/CoDAMoS_Extended.xml"
         type="ONTOLOGY" /> </tnsa:Category1>
   <tnsa:Category2> <tnsa:Constant type="STRING">
       GREDIA_RELATIVE_TIME</tnsa:Constant> </tnsa:Category2>
  ...<tnsa:Operand2><tnsa:Constant type="STRING">
      SECONDS-5</tnsa:Constant></tnsa:Operand2></tnsa:Condition>
</tnsa:LogicalExpression></tnsa:ConstraintQuery
```

Fig 6: Example of contextual constraint for query Q1

Figure 6 shows an example of a soft contextual constraint (C3) for query Q1 about the payment processing time. This constraint specifies that any candidate payment service, i.e., services that match operation *transferAmount*, needs to have a context operation classified in the category GREDIA_RELATIVE_TIME in the ontology *http://eg.org/CoDAMoS_Extended.xml*, and the result of executing this operation has to be less or equal to SECONDS-5 for the service to be considered.

Arithmetic expressions define computations over the values of elements or attributes in service specifications or context information. They are defined as a sequence of arithmetic operands or other nested arithmetic expressions connected by arithmetic operators (plus, minus, multiply, and divide operators). The operands can be con-

textual, non-contextual, constants, or functions.

A *function* supports the execution of complex computation over a series of arguments. The results of these computations are numerical values that can be used as an operand in an arithmetic expression. The schema for arithmetic expressions and functions is available at [16].

4 Service Discovery Process

The service discovery process realized in RSDF can execute queries in pull or push mode. The pull mode of query execution is performed to identify services (i) that are initially bound to a service-based application and their replacement candidate services, (ii) as a first step in the push mode of query execution, (iii) due to changes in the context of an application environment, or (iv) when a client application requests a service to be discovered. The push mode is performed when a service in an application needs to be replaced due to any of cases (a)-(d) described in Sec. 1. The matching process followed in the pull and push mode of query execution is described below.

4.1 Query Matching Process

Matching between queries and services is executed in two stages: the *filtering* stage and the *ranking* stage.

In *filtering* stage, only the hard non-contextual constraints of a query are evaluated against service specifications and the candidate services that comply with these constraints are identified. The reason for filtering out all the services that do not satisfy hard non-contextual constraints is to optimise subsequent computations.

The *ranking* stage and has three substages. In the first of these substages, the structural and behavioural parts of a query are evaluated against the services maintained by the filtering stage and a *structural-behavioural partial distance* between each of these services and the query is computed. In the second substage, the soft non-contextual constraints of the query are evaluated against each candidate service and a *soft non-contextual partial distance* is computed for each such service. Finally, in the third substage, the contextual constraints of the query are evaluated against the candidate services and a *contextual partial distance* is computed for each candidate service. At the end of the ranking stage, the partial distances computed for each service are aggregated into an overall distance and only services whose distance to the query is below a certain threshold are maintained. The distance threshold is specified in the query as the value of the query element *Parameter* and a default threshold of 0.5 is used if the query does not specify a threshold.

It should be noted that a query is executed by the framework only if it contains a structural sub-query. This is necessary, as services cannot be identified for a running application unless their interface is known. All other parts of a query, however, can be omitted and if they are, the respective stages in matching are not executed.

The overall distance between a service S and a query Q is computed according to the following formula:

$$OD(Q,S) = (w_i * D_{Str\_Beh}(Q,S) + w_j * D_{NCC}(Q,S) + w_k * D_{CC}(Q,S)) / (w_i + w_j + w_k)$$

In this formula,

- $D_{Str\_Beh}(Q,S)$ is the structural_behavioural partial distance between a query and a service;
- $D_{NCC}(Q,S)$ is the soft non-contextual constraint partial distance between a query and a service;
- $D_{CC}(Q,S)$ is the contextual constraint partial distance between a query and a service; and
- w_i, w_j, w_k are weights with values between [0, 1] representing different priorities for the various partial distances.

Structural and Behavioural Matching

The structural and behavioural evaluation of a query against services is executed by comparing operations in the structural sub-query with operations in the WSDL specifications of services. Following this, the behavioural part of a query is matched with the BPEL (behavioural) specifications of services. In this process, a match between a service and a query is found only if for each operation in the query (Qop), the service has an operation which has the same name as Qop, input parameters whose data types are supertypes of the types of the input parameters of Qop, an output parameter whose type is a subtype of Qop's output. This is because when these conditions hold, the input information assumed for invoking Qop will cover the input information needed by Sop and the output information produced by Sop will cover the output information expected by Qop. Furthermore, the service must have a behavioural model satisfying all the behavioural conditions of the query. Given that the above conditions are satisfied by a service S, a structural_behavioural distance between it and the query Q ($D_{Str\_Beh}(Q,S)$) is also computed to enable the ranking of S with respect to other services that satisfy the same conditions. This distance is computed by the formula:

$$D_{Str\_Beh}(Q,S) = MIN(\sum_{1 \le i \le n \wedge 1 \le j \le m} d_{SB}(Qop_i, Sop_j) / n)$$

where:

- $d_{SB}(Qop_i, Sop_j) = (d_S(Qop_i, Sop_j) + d_B(Qop_i, Sop_j)) / 2$;
- n is the number of operations in query Q;
- m is the number of operations in service S;
- $d_S(Qop_i, Sop_j)$ is the structural distance between an operation in a query and an operation in a service; and
- $d_B(Qop_i, Sop_j)$ is the behavioural distance between an operation in a query and an operation in a service.

The structural distance between query and service operations ($d_S(Qop_i, Sop_j)$) is calculated by matching the signatures of service and query operations. This matching is based on a comparison of the names of the query and service operations (the names must be the same in order to find a successful match) and the graphs representing the data types of the input and output parameters of the operations, and the names of operations and parameters.

The comparison of the input/output parameter data type graphs is based on a variant of the *VF2 algorithm* that we have developed to detect morphisms between service graphs and if a graph is a subgraph of another [60]. This

variant allows matches between data type graph edges whose names have a synonym in WordNet, their origin/destination nodes have matching incoming/outgoing edges. Based on this algorithm, a query operation Qop with input and output parameter data type graphs IG_{Qop} and OG_{Qop} matches a service operation Sop with input and output parameter data type graphs IG_{Sop} and OG_{Qop}, if IG_{Sop} is a sub-graph of IG_{Qop} and OG_{Sop} is a super-graph of OG_{Qop}. This criterion guarantees that the data types of the input parameters of Sop are supertypes of the data types of the input parameters of Qop and the output of Sop has a type that is a subtype of Qop's output.

Given the morphism m_{in} from IG_{Sop} to IG_{Qop} and the morphism m_{out} from OG_{Qop} to OG_{Sop}, the structural distance between Qop and Sop is computed by the formula:

$$d_S(Qop,Sop)=(w_i*d_{Ling}(Qop,Sop)+w_j*d_{in}(Qop,Sop)+ w_k*d_{out}(Qop,Sop))/(w_i+w_j+w_k)$$

where:

- $d_{Ling}(Qop,Sop)$ is the sum of the linguistic distance of the names of operations and parameters based on WordNet [37];
- $d_{in}(Qop,Sop)=|NMapEd(IG_{Qop},IG_{Sop})|/|Ed(IG_{Qop},IG_{Sop})|$;
- $d_{out}(Qop,Sop)=|NMapEd(OG_{Qop},OG_{Sop})|/|Ed(OG_{Qop},OG_{So}|$;
- $NMapEd(G1,G2)$ is the set of the non mapped edges of the graphs G1 and G2;
- $Ed(G1,G2)$ is the set of all the edges of the data type graphs G1 and G2; and
- w_i,w_j,w_k are weights with values between [0,1]

To illustrate the computation of the structural distance, consider a query operation *Qop': Search(news:News):string*. The input parameter *news* of this operation has a composite data type *News* consisting of the attributes *topic:string*, *period:date*, and *area:string*. Consider also a service operation *Sop':Search(searchnews:News):string* with an input parameter, called *searchnews,* of a composite type *News* consisting of the attributes *subject:string*, *period:date*, *region:string*, and *size:integer*. Figure 7 shows the data type graphs for the input and output parameters of the query and service operations and the mapping that the matching process of RSDF has detected between these graphs. The structural distance between Qop' and Sop' is $d_S(Qop',Sop') = (0.5+0.25+0)/3 = 0.25$, since

- $d_{Ling}(Qop',Sop') = 0.5$ (i.e., the aggregate distance between strings "Search" and "Search"; and between strings "news" and "searchnews")
- $d_{in}(Qop',Sop') = ¼ = 0.25$ (the edge *size* in the graph of Sop' has no matching edge in the graph of Qop')
- $d_{out}(Qop',Sop') = 0$ (the output parameters in Qop' and Sop' are both of type *string*).

In RSDF, structural distances are computed for each possible pair of an operation in a query Q and an operation in a service S.

After computing the structural distance for each pair of query and service operations, RSDF identifies all the possible mappings between the operations in Q and operations in S in which each operation in Q (Qop_i) is mapped onto a single operation in S (Sop_j). For example, given two query operations (Q_{opA}, Q_{opB}) and two service operations (S_{opA}, S_{opB}), the following combinations of mappings would be examined:

{Q_{opA}:S_{opA}, Q_{opB}:S_{opA}}, {Q_{opA}:S_{opA}, Q_{opB}:S_{opB}},
{Q_{opA}:S_{opB}, Q_{opB}:S_{opA}}, {Q_{opA}:S_{opB}, Q_{opB}:S_{opB}}

For each of these mappings, RSDF computes the behavioural distance between the mapped service and query operations ($d_B(Qop_i, Sop_j)$). This distance is calculated based on comparisons of paths representing the behavioural sub-query and behavioural service specification. The behavioural distance between Qop and Sop is computed as:

- $d_S(Qop,Sop)=0$, when the path of the behavioural subquery that contains Qop can be matched with a path in the state machine of a service;
- $d_S(Qop,Sop)=1$, otherwise.

More specifically, the behaviour matching is executed by transforming the BPEL behavioural specifications of each service into a state machine SSM and the behavioural sub-query into another state machine QSM, and verifying if each path in QSM can be matched with a path in SSM. The transformation of BPEL specifications into state machines is described below.

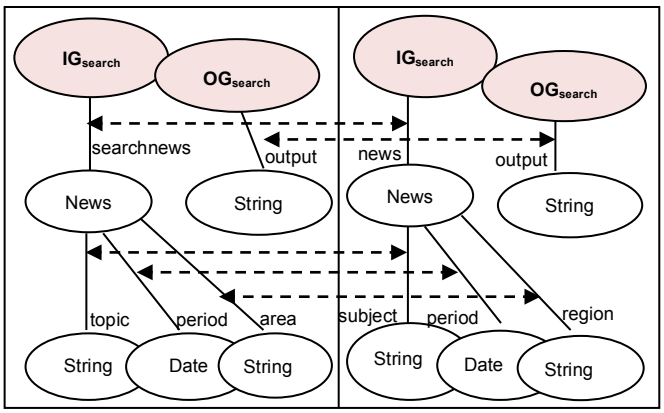

Fig 7: Example of data type graphs

When a path representing the behavioural sub-query can be matched with a path in the state machine of a service, the behavioural distance for each pair of mappings of query and service operations in these paths is set to zero (i.e., $d_B(Qop_i, Sop_j) = 0$). Otherwise, the behavioural distance for each pair of mappings of query and service operations in these paths is set to one ($d_B(Qop_i, Sop_j) = 1$). As an example, consider an operation Qop1 defined as a *GuaranteedMember* condition in the behavioural sub-query of Q and mapped to a service operation Sop1 in one of the possible operation mappings. This condition is satisfied by the state machine of S, if Sop1 exists in all possible paths of this state machine. In this case, $d_B(Qop_1, Sop_1) = 0$. Otherwise, $d_B(Qop_1, Sop_1) = 1$.

As another example, consider a *Sequence* condition in the behavioural sub-query Q with operations Qop2, Qop3 and Qop4. Suppose that Qop2, Qop3, and Qop4 are mapped to service operations Sop2, Sop3, and Sop4, respectively in one of the operation mappings. The above

Sequence condition is satisfied by the state machine of service S, if Sop2, Sop3, and Sop4 exist in this order in a path of the state machine. In this case, $d_B(Qop_2,Sop_2)=d_B(Qop_3,Sop_3)=d_B(Qop_4,Sop_4)=0$. If there is no path with Sop2, Sop3, and Sop4 appearing in this order, $d_B(Qop_2,Sop_2)=d_B(Qop_3,Sop_3)=d_B(Qop_4,Sop_4)=1$.

After computing the structural and behavioural distances for all pairs of query and service operations in all possible operation mappings and calculating $d_{SB}(Qop_i,Sop_j)$ for each pair of operations, the framework selects the mapping that has the minimal value for the sum of $d_{SB}(Qop_i,Sop_j)$ for all the pairs of query and service operations, divided by the number of operations in the query, as specified by the formula for $D_{Str\_Beh}(Q,S)$ above.

In order to illustrate these computations, consider a simple query Q2 with two required operations:

- *credit(accountId:string, amount:double):balance*
- *debit(accountId:string, amount:double):balance*

and a behavioural condition stating that the operation *credit* needs to be executed before the operation *debit*.

Consider also a service $S_{BankNew}$ having the three operations below and the state machine shown in Figure 8.

- *Op1 = credit(accountId:string, amount:double):balance*
- *Op2 = debit(accountId:string, amount:double):balance*
- *Op3 = getBalance(acountId:string, date-time:double)*

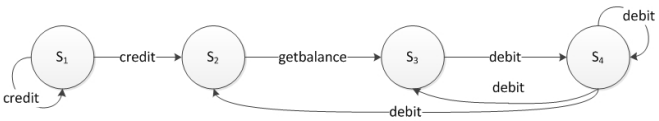

Fig 8: State Machine of service $S_{BankNew}$

For this example there are 49 possible combinations of mappings of operations in Q2 and transitions in the state machine of $S_{BankNew}$. Figure 9 shows some of these combinations of mappings. In the figure, combinations C1 to C8 have structural_behavioural distance (D_{SB}) equal to zero, and combinations C9 and C10 of mappings have structural_behavioural distance not equal to zero. The other combinations are not shown here due to space limitations.

C1={credit:(S1→S2), debit:(S3→S4), D_{SB} =0.0};
C2={credit:(S1→S2), debit:(S4→S4), D_{SB} =0.0};
C3={credit:(S1→S2), debit:(S4→S2), D_{SB} =0.0};
C4={credit:(S1→S2), debit:(S4→S1), D_{SB} =0.0};
C5={credit:(S1→S1), debit:(S3→S4), D_{SB} =0.0};
C6={credit:(S1→S1), debit:(S4→S4), D_{SB} =0.0};
C7={credit:(S1→S1), debit:(S4→S2), D_{SB} =0.0};
C8={credit:(S1→S1), debit:(S4→S1), D_{SB} =0.0};
C9={credit:(S1→S2), debit:(S2→S3), D_{SB}=0.0834};
C10={credit:(S3→S4),debit:(S2→S3), D_{SB} =0.67};

Fig 9: Structural_behavioural distances for all combinations

The behavioural distances for the mappings in C1 to C9 are zero, since in these combinations the mapping of the query operation onto state transitions guarantee the order specified by the behavioural condition in the query. More specifically, in C1, the query operation *credit* is mapped to service operation labelling the transition S1→S2, the query operation *debit* is mapped to service operation labelling the transition S3→S4, and S1→S2 occurs before S3→S4. Similar situations occur in combinations C2-C8, and C9 (in them *credit* is mapped to S1→S2, *debit* is mapped to S2→S3, and S1→S2 occurs before S3→S4). In C10, however, the behavioural distance is set to 1 since the transitions mapped to *credit* and *debit* query operations do not preserve the order specified by the behavioural condition of the query. For example, in C10 *credit* is mapped to transition S3→S4 that is labeled by a service operation that occurs after the transition S2→S3 which is labeled by the service operation to which query operation *debit* is mapped.

The calculation of the *structural_behavioural distance* for each mapping is computed by the formula $d_{SB} = (d_S + d_B)/2$. The structural_behavioural distance (D_{SB}) for each combination is taken as the total of all the d_{SB} values in the mappings in that combination divided by the number of query operations.

Soft Non-Contextual Constraint Matching

The evaluation of soft non-contextual constraints is executed by evaluating constraint expressions in the constraint sub-queries against service specification facets (see Sec. 3). This evaluation takes place by retrieving the values of the XPath expressions from service specification facets and evaluating the arithmetic, relational and logical expressions that define the constraint using these values. The result of this evaluation is a binary value indicating whether the constraint is satisfied (1) or not (0). Based on the evaluation of individual constraints, a soft *non-contextual constraint partial distance* ($D_{NCC}(Q,S)$) is also calculated between a query Q and each service S. The calculation of this distance is based on the function:

$$D_{NCC}(Q,S) = \sum_i w_i * D(C_i) / \sum w_i$$

where

- C_i is a soft non-contextual constraint in Q ($1 \le i \le n$; n is the number of soft non-contextual constraints in Q);
- w_i is the weight in [0,1] indicating the significance of the constraint C_i in Q; and
- $D(C_i)$ equals 0 when C_i is satisfied by service S and 1 when C_i is not satisfied by service S

Contextual Constraint Matching

The evaluation of contextual constraints is based on the approach described in [49]. This approach assumes that a service or the individual operations of it may have one or more context operations associated with them that can be invoked at runtime to provide information that changes dynamically and frequently. Following this approach, the contextual constraints in *SerDiQueL* are specified as logical combinations of conditions over the return values of context operations of services. Then, during the evaluation of contextual constraints, RSDF identifies the relevant context operations, invokes them, and uses their return values to check if the conditions of the context constraint are satisfied or not.

The context operations associated with a service are specified by *context facets*. More specifically, a context fac-

et specifies the context operations of a service by listing the operation and its semantic category. This category is defined in reference to some context ontology. Whilst the description of categories in RSDF can be based on different types of ontologies (as long as they are specified in XML), in the current implementation of RSDF, we have used an extended version of the CODAMOS ontology [9].

An example of a context facet is shown in Figure 10. As shown in the facet, the service operation *transferAmount* is associated with the context operation *getTime* whose semantic category is GREDIA_RELATIVE_TIME in the CODAMOS ontology. This facet will be used whilst evaluating the contextual constraint C3 of query Q1 (see Figure 6). In particular, RSDF will check the facet of service "21764851280153632" to see if it has a context operation with the same semantic category as the one specified in the constraint sub-query (i.e., GREDIA_RELATIVE_TIME). In this case, the context operation *getTime* has this category. Thus, RSDF will invoke it and use its return value to evaluate the condition of C3.

```
<LanguageSpecificSpecification>
  <FacetType>Context</FacetType> . . .
    <FacetSpecificationData>
      <serviceOperationContexts service="21764851280153632">
       <serviceOperation>transferAmount</serviceOperation>
       <context>
        <contextServiceOperation>
         <contextServiceId>2176</contextServiceId>
         <contextOperationName>getTime</contextOperationName>
        </contextServiceOperation>
        <timeValidity> <validTime>5</validTime> <unit>minutes</unit>
        </timeValidity>
        <contextOperationCategory>
         <ontology>http://…/CoDAMoS_Extended.xml</ontology>
         <categoryExpression>
          /rdf:RDF/owl:Class[@rdf:ID='GREDIA_RELATIVE_TIME']
         </categoryExpression>
        </contextOperationCategory>
       </context>
    </serviceOperationContexts>...</FacetSpecificationData>
</LanguageSpecificSpecification>
```

Fig 10: Example of context facet

The evaluation of each contextual constraint results in a binary value indicating whether the constraint is satisfied or not and the computation of the *contextual constraint partial distance* between a query Q and a service S ($D_{CC}(Q,S)$) is evaluated using the same formula as in the case of non-context constraints (see formula for D_{NCC}).

Transformation of BPEL into State Machines

The state machine that is used by the RSDF framework is based on transition tuples (t-tuples) specified as

t(service, initial-state, action, destination-state)

where

- *service* is the logical name of the service whose behaviour is specified by the state machine;
- *initial-state* is the state from which the transition originates;
- *action* is the action that triggers the transition;
- *destination-state* is the state to which the transition will result.

The above representation denotes that if in the initial-state the service becomes aware of an event requesting the execution of the action associated with the transition, it will execute the action and move to the destination-state. The action associated with a transition may be of one of the following types:

- *receive action*: a communication action signifying that the service has received a message requesting the execution of an operation.
- *send action*: a communication action signifying that the service sends a message notifying the results of the execution of an operation.
- τ *action*: an internal action undertaken by the service that cannot be interpreted by an external service partner (e.g., an automatic transition which is not triggered by any event or a call to a third party operation).
- *assign action*: an action that assigns a value to a service variable.
- *cond action*: an action which checks if a condition associated with a transition is satisfied in order to allow the transition to take place.
- *after action*: an action that forces the service to undertake a transition after a specific time period following the time at which it arrived at the initial-state.

Based on the above representation, the main elements of a BPEL specification are transformed into state machines as discussed below.

- *Invoke/receive:* Invoke and receive are BPEL activities which invoke an operation in a partner link (service) of a BPEL process and receive a message requesting the execution of an operation, respectively. These constructs are mapped to transitions triggered by send and receive actions, respectively.
- *Pick/onMessage/onAlarm:* This activity contains an ordered list of one or more event and activity pair. Pick makes a BPEL process wait for the occurrence of one of these events and then perform the activity associated with it as soon as it occurs. A pick may define two types of events: (i) message events, which signify the arrival of a message; and (ii) alarm events that set a timer. This activity is translated into a path of a state machine that has one state *s* immediately before pick and multiple transitions triggered by event receive actions originating from *s*. Each of these transitions represents the different message that they may receive and is followed by a transition representing the activity that follows the receipt of the event. The timeout onAlarm is mapped to a transition triggered by an after action.
- *Flow:* This activity in BPEL specifies a set of paths in the process that should be executed concurrently. The state machine that is generated for flow activities contains concurrent transition paths representing the full graph of possible sequences of transitions when the BPEL process takes a new step in one parallel partition.
- *Switch:* This activity in BPEL specifies an ordered list of one or more conditional branches that include other activities. The conditional branches are considered in sequential order and the activity(ies) of the first branch whose condition becomes true is executed. In the case where no

condition holds true, a default branch can be specified. This activity is mapped to a number of transitions from a current state triggered by cond actions, which specify the condition of the particular branch of execution.

▪ *While:* This BPEL activity is used to specify that a group of other activities will be executed iteratively for as long a condition associated with the activity remains true. A while activity is translated into a fragment in a state machine that has a choice state s0 preceding the while test and two outgoing transitions triggered by cond actions. The first of these transitions corresponds to the case where the condition c of the loop is satisfied and has the form < _, *s0,cond(c), b0*> where b0 is the state from which the first activity of the loop can be executed. The second transition has the form < _, *s0, !cond(c), sn*> where sn signifies the state of the process after the execution of the loop. The body of the loop is a sub-machine that has one initial state b0 and one final state sn-1 that is the origin of transition t to the original state of the loop, which is automatically triggered after *sn-1* is reached (we call t an "automatic" transition). This transition signifies the move to the state where the satisfiability of the condition of the While activity has to be checked again.

The transformation approach used in the framework has some limitations. These include the inability to create state machine representations for *faultHandlers* and *link* constructs in BPEL processes. There are also limitations in processing WSDL specifications accompanying a BPEL process. In particular, XSD Schema import statements are currently not supported and prefix names for namespaces have to be unique across the WSDL specification.

Discussion

The matching process implemented by RSDF accommodates a certain degree of flexibility whilst ensuring that any returned service operation can replace, at least at an interface level, the service operation for which it was discovered. This is because the matching process guarantees that the types of the input and output parameters of the new operation are subtypes and supertypes of the types of the input and output parameters of the operation that will be replaced, respectively. Furthermore, all the returned service operations are guaranteed to satisfy the behavioural conditions and the hard constraints set in the query. Hence it is guaranteed that the service-based application that will use the new operation will be able to provide the data required for its invocation (as it was already able to use the previous operation). It will also be able to accept the data produced by the new operation that corresponded to the structure of the output types of the old operation. The framework also produces a mapping between the types of the parameters of the old and new operations to enable developers understand (offline) the structure of the new operation.

Currently, RSDF assumes that the behaviour of services is described in BPEL within service registries and, hence, when a SerDiQueL query is evaluated, it retrieves and translates such descriptions into a state machine and then checks if the behavioural conditions in a SerDiQueL query are satisfied by this state machine. It should be noted, however, that other languages for describing service behaviour in service registries could be supported (e.g., UML), as long as the descriptions of service behaviour that are expressible in these languages could be translated into a state machine.

It should also be noted that although our framework can support discovery based on different types of service descriptions, its approach is modular and can work even for services with incomplete faceted descriptions. More specifically, the minimal requirement for a service is to have a structural (WSDL) description and, in such cases, the framework executes only the structural part of queries and computes only the structural distances. Then subject to the availability of further facets in a service description (e.g. BPEL, QoS descriptions, context providing operations), the framework will perform the additional types of matching that it supports.

This modularity is necessary for coping with cases of incomplete service descriptions. Nevertheless, as discovery takes place at runtime the framework needs to ensure that any discovered services will be usable in the system at the interface level. Hence, it only considers services that have a structural description and always performs structural matching.

Our approach does not use formal ontologies to support the structural and behavioural matching. Instead, it uses WordNet. The use of WorldNet gives matching some flexibility when names of operations, parameters and attributes of parameter types are not exactly the same. Although WordNet is not a formal ontology and does not include axiomatic specifications of concepts, it includes semantic relationships between concepts represented by words (e.g., synonyms, part of relations). The use of WordNet does not require the annotation of service specifications with ontologies and the creation of the dictionary/ontology itself. This is the main reason for not using formal ontologies for structural/behavioural matching.

4.2 Pull and Push Query Execution

In the pull mode of query execution, the service requester of RSDF (see Sec. 2.2) invokes the service matchmaker to execute a query. The service matchmaker executes the query and maintains services whose distance from the query does not exceed a specific threshold. The set of maintained services is sorted in ascending distance order and returned to the client application for further action.

In the push mode of query execution, the client application subscribes to RSDF the services it deploys and a query Q for each of them. Based the subscribed query for each service *S*, RSDF initially retrieves a set of services *Set_S* that could replace *S* (if necessary) by executing the query as in the pull mode and, subsequently, maintains an up-to-date version of *Set_S* as changes in the descriptions and context of the services and/or their application's environment are notified to it. *Set_S* includes only services whose overall distance from the query sub-

scribed for S does not exceed a given threshold and is sorted in ascending distance order.

It should be noted, however, that although *Set_S* includes candidates that could replace S, the replacement of *S* in the application does not take place right after the first or subsequent modifications of *Set_S*. This is because an immediate replacement might be inappropriate. In cases, for example, where the service *S* is executing some transaction on behalf of the application, at the time when a new better service is found, no replacement should take place. In RSDF, the decision to stop the execution of the application in order to replace a service for which a better alternative service has been found is based on *replacement policies*. Policies are associated with the different functional roles that are assumed by services in the application and specify if the service that is currently bound to a role should be replaced immediately when a better service is found, after the termination of a specified computation, or after the application terminates.

The push mode service discovery process that maintains the set *Set_S* of candidate replacement service for a given service *S* covers four different cases. These are cases where: (a) *S* becomes malfunctioning or unavailable (*Case A*); (b) there are changes in the structure, functionality, quality or context of any service in *Set_S* or *S (Case B)*; (c) there are changes in the context of the application environment (*Case C*); or (d) new services become available or existing services have their characteristics modified (*Case D*). In the following, we discuss the push execution mode for each of these cases.

Case A: In this case, the service *S* is replaced by the first service S′ in *Set_S*. By virtue of the process of maintaining this set, S′ is guaranteed to have the smallest distance to query Q associated with *S*. Following the replacement, S′ is removed from *Set_S*.

Case B: Suppose S′ is a service that is currently bound to the application or another service in its associate replacement *Set_S*. This case arises when new versions of the structural, functional, quality, or context facets of S′ become available in a service registry. When a change in some characteristic of S′ occurs, the new versions of the changed facets or service context information are evaluated against query Q to verify if S′ still matches the query. The new overall distance between Q and S′ is also calculated. If S′ is a candidate replacement service in *Set_S*, it remains in it only if the new distance between S′ and Q is below the threshold distance. Also the relevant position of S′ in *Set_S* might change due to the new distance. Following this, if S′ becomes the best replacement service in *Set*_S, S′ will replace S when the relevant replacement policy allows it. If S′ is the service currently deployed by the application, but is no longer the best option according to its new distance from Q, it will be replaced by the first service in *Set_S* as soon as the replacement policy permits the change. Furthermore, if the new distance between S′ and Q makes S′ a non eligible member of *Set_S*, *S′* will be removed from *Set_S* and its subscription will be removed from RSDF. Also a new replacement service for S′ in *Set_S* will be located.

Case C: In this case, a value in a context constraint in query *Q* is modified and a new query Q′ needs to be created to reflect the new context value. The service S that is currently bound to the application needs to be evaluated against the new context constraint in Q′. If *S* does not match the new query *Q′*, the services in *Set_S* will be evaluated against Q′ and a new version of *Set_S* may be generated. This is necessary for identifying the service S′ in *Set_S* with the best fit to Q′ and bound it to the application so that it can continue its execution, whilst trying to find new services that match Q′ from the service registries. Note that S′ might not have the best fit with *Q′* given all available services in the registry. However, the use of the best service *S′* in the current *Set_S* in this case is acceptable as it will allow the application to continue. Moreover, the context constraints are soft constraints used for ranking services with respect to queries, rather than filtering them out. Following the use of S′, RSDF will do an exhaustive search in registries (pull mode) to update *Set_S* based on *Q′*. The same exhaustive search will be used if no service in the current Set_S matches Q′. Following, the update of *Set_S*, if a new service in it is better than S′, it will replace S′ subject to the replacement policy.

Case D: This case arises when new services appear in registries for the first time or descriptions of existing services in registries that were not matching a query Q before change (the latter services are not covered by Case C since, as they are not members of *Set_S*, the changes in their characteristics will not be notified to RSDF through the existing subscriptions for *Set_S*). Once RSDF is notified of new or updated service descriptions, it evaluates them against query Q for each service S deployed in the application. Depending on the result of this evaluation, the new/updated service may become member of *Set_S* or even replace S in the application, subject to the criteria of the replacement policy for S.

In the approach, the replacement policy used in Cases (A)-(D) described above takes into consideration the position of a service S that may need to be replaced with respect to the current execution point of the service-based application. More specifically, the replacement policy considers the cases in which changes need to be performed so that the application can continue its operations; changes can wait to be performed after the current execution of the application; and no changes are required. For a replacement policy, the approach considers three different positions, namely:

(i) *not_in_path*: when service S in not in the current execution path of the application; S appears in a different branch of the application's execution path or before the current point in the execution path;
(ii) *current*: when service S is in the current execution point of the application;
(iii) *next_in_path*: when service S is in the current execution path of the application, and will be invoked some time in the future.

When the position of a service S to be replaced is

not_in_path, S is marked for replacement when S is accessed in a future execution of the application; when the position of S is *current*, S needs to be replaced; when the position of S is *next_in_path*, S is marked to be replaced when S is accessed in the current execution.

It should be noted that notifications about changes in services S′ falling in Case B are dealt with according to their priority. More specifically, the service matchmaker maintains three notification queues – a high, medium and low priority queue – and notifications for services bound to the application are placed at the end of the high priority queue, notifications for services in some replacement set (Set_S) are placed in the medium priority queue, and notifications for other services are placed in the low priority queue (i.e., the queue of new services). Also, the services with notifications in the high and medium priority queues are marked as "unsafe" in order to prevent the application from using them before the notifications that have arrived for them are processed. Furthermore, if a notification N′ arrives for a service for which there is already an earlier notification N in the same queue which has not been processed yet, the facets identified in N′ are added to those of N in the queue rather than appending N′ to the end of the queue. Hence, all the unprocessed notifications of a service are merged.

Matchmaker processes notifications from lower priority queues only if there are no notifications in higher priority queues. This heuristic ordering of processing ensures that: (a) notifications regarding the services which are currently bound to the application and are the most critical for it are processed first, (b) notifications for candidate replacement services which will enable timely operational replacement of services will be processed next, and (c) notifications for new services that can only lead to optimizations will be processed last. This is a measure for dealing with high frequency service emergence and/or update rates, which can stress the resources of RSDF.

An approach for executing changes in a service-based application can be performed by stopping the system, making the necessary changes, and resuming the system [2]. Other approaches use binding partner links during execution time of the system [23]; proxy services as placeholders for the services in a composition, instead of having concrete services referenced in the system [3][27]; or even an adaptation layer based on aspect oriented programming with information about alternative services [36]. In this framework, we use proxy services to support changes in the service-based application during execution time and, therefore, avoiding changes in the original application specification. More specifically, RSDF maintains a record associating the logical references to services within an application with pointers to the actual services used and when a call is made by the application the logical reference is resolved to the actual endpoint where the service can be called.

5 Evaluation

To evaluate RSFD, we have performed a set of experiments whose objective was to measure and analyse the performance of both pull and push modes of query execution given queries incorporating structural, behavioural, non-contextual, and contextual conditions.

This evaluation has not focused on other criteria for assessing information retrieval techniques (as runtime service discovery), notably the recall and precision of the retrieval (discovery) algorithm. The reason for not focusing on such criteria is that the matching algorithms which are deployed by our approach (i.e., structural, behavioural and constraint matching) ensure that the services returned by a query always satisfy the minimum set of conditions, which are necessary for being able to be used as substitutes of services already deployed by a system.

In particular, as we discussed in Sect. 4.1, the service operations which are returned by a query are guaranteed to have the same name with the operations required by a query, and input/output parameters whose types are subtypes/supertypes of the types of the input/output parameters of the query operations. Hence, no inaccuracies that prevent substitutability may arise at the interface level. Furthermore, all the returned service operations are guaranteed to satisfy the behavioural conditions and the hard constraints set in the query. Hence, the only possibility of an inaccuracy in query results is to use a query, which does not specify correctly the conditions about the interface, behavior and other characteristics of acceptable services. Evaluating whether the discovery queries are correctly specified would be beyond the technical core of the discovery framework, which is the focus of this paper.

5.1 Experimental Setup

In the experiments, we used a registry of 60 services. Each service was described in terms of a structural (WSDL), behavioural (BPEL), quality (XML-based), and context (XML-based) facet, with a total of 240 service facets in the registry. The structural specifications of the 60 services had a mixture of four, five, and six operations with a total of 300 operations for all the 60 services. The complexity of the operations varied with operations containing one, two, or three input parameters, and all operations with one output parameter.

The services used in the experiments had been collectively selected and built by the industrial partners in the GREDIA project [19]. The services came from different service providers and were concerned with different domains including: (a) online banking, (b) online media channels, (c) online retailing, (d) Internet searching, and (e) travel planning and booking.

The evaluation was incremental using three different subsets of the registry having 20, 40 and 60 services, respectively. The incremental evaluation was adopted in order to analyse whether the increase in the number of services affects the query execution time. In the case of query execution in push mode, we also used threshold values to guarantee that the set of up-to-date candidate services always contained ten services. The execution time of each query was calculated as the average across

five different executions of it using a Pentium 1.2 GHz machine with 1.24 GB RAM.

Table 1: Types of queries used in the experiment

Q1	Structural
Q2	Structural and behavioural
Q3	Structural, behavioural, and soft non-contextual constraint
Q4	Structural, behavioural, soft non-contextual, and contextual constraints

In the experiments, we used four different queries from the "*on-the-go-News*" application scenario that were created as variants of query Q1 presented in Figures 3, 4, 5, and 6 without hard constraint. The queries included different types of discovery criteria, which are summarized in Table 1. More specifically, the queries used in the experiment were composed of a structural sub-query with the WSDL specification of $S_{Payment}$ (see Figure 3) with six operations; a behavioural sub-query with the conditions shown in Figure 4 and described in Sect. 3.2; a soft non-contextual constraint as shown in Figure 5; and a soft contextual constraint as shown in Figure 6. Hard constraints were not used in the queries since they could filter out services before ranking and, therefore, artificially reduce the query execution time. In the experiments, we considered the weights associated with the partial distances in the overall distance function and the weights in the structural distance (see Sect. 4.1) with values 1.

5.2 Performance Results

Table 2 and 3 summarise the results of the evaluation. In particular, Table 2 presents the execution time of different queries in pull mode and the average time required for retrieving services from the registry, for different sizes of registries. Table 3 presents a breakdown of the total query execution time into the time required for retrieving services from the registry, and structural, behavioural soft non-context constraint and context constraint matching of query Q4. All times in Tables 2 and 3 are in milliseconds. Furthermore, for the results in Tables 2 and 3, all the services in a registry of a given size *n* were evaluated against all the criteria of each query in order to ensure that the evaluation time measured for different types of criteria was not affected by the order of criteria evaluation.

Table 2: Pull mode execution times for the different queries (in msec)

# of Services	20	40	60
Registry Retrieval	14025	24615	35170
Q1	763	1392	2057
Q2	14828	27431	39936
Q3	15022	27781	40464
Q4	22547	42768	63924

As shown in Tables 2 and 3, the time to retrieve services from the registry was substantial in contrast to the time taken to execute the different matchings. This is because the eXist database [17] that was used to implement the registry has a low data retrieval performance. The use of the proactive push query execution mode of RSDF alleviated this problem as replacement services are selected in parallel to the execution of an application from an up-to-date set of candidate services, as discussed below and shown in Table 4. Moreover, except in the case of changes in the context of an application environment, the set of candidate services has a reduced number of services when compared to an entire service registry.

Table 2 shows that the total execution time for all the different queries increased linearly with the addition of more services in the registry. Table 3 shows that the execution time for different types of matching criteria also increased linearly as the number of services in the registry increased in all cases. The experiment also showed that the time required for behavioural matching was substantially higher than the time required for the other types of matching. This is because in behavioural matching, a behaviour path in a query needs to be evaluated against all the paths in the state machine of each service, and all possible combinations of mappings between query and service operations need to be considered.

Table 3: Pull mode execution times for each matching criteria and registry retrieval (in msec)

# of Services	20	40	60
Registry retrieval	14025	24615	35170
Structural matching	763	1392	2057
Behavioural matching	14065	26039	37879
Non-Context constraint matching	194	350	528
Context constraint matching	7525	14987	23459
Total	36572	67383	99093

As shown in Table 3, the time required for non-contextual constraint matching was smaller than the time required for each of the other types of matching. This time was also significantly lower than the time required for matching contextual constraints. This was because in non-contextual constraint matching, the non-contextual condition in a query is evaluated against facets in the registry by comparing elements retrieved by evaluating XPath expressions. In the case of contextual matching, however, the computation is more expensive as it requires the invocation of context operations at runtime in order to obtain the context values for evaluating the context conditions in queries.

Table 4 presents the results of the push mode execution of query Q4 including the time needed to: (a) prepare the set of candidate services for a subscribed query (i.e., *Set_S*) at the initial stage in the process, as discussed in Sec. 4.2 and (b) identify a new service for replacing a service S in the service-based application due to (i) unavailability of S, (ii) availability of a new service, or (iii) changes in the service bound to the application. The table presents the time required for executing Q4 in five different times for each of the cases (i), (ii) and (iii) (i.e., runs R1 to R5) and the average time across the five runs in each case. The events for cases (i), (ii) and (iii) were created by simulation.

As shown in Table 4, the time of identifying a service in cases (i), (ii), and (iii) is very small in contrast with the time of identifying a service in pull query execution mode

(compare the values in Total row in Table 3 for 20, 40, and 60 services in the registry and the values in the Avg column in Table 4). Also, the initial time required for building the set of candidate services *Set_S* for a given service and query in the push execution mode is comparable to the time needed for executing a query in pull mode (compare the time values for the part of Table 4 concerned with "Prepare candidate services" with the respective values in Table 3) . It should be appreciated that in the case of push mode, the initial phase for building *Set_S* is performed only once for a given service and query and in parallel to the execution of the application. Thus, the time needed for identifying a service in cases (i), (ii) and (iii) are the ones shown in the last three rows of Table 4.

Table 4: Times for executing Q4 in push mode of execution (in msec)

	# Ser.	Reg. Retrieval	Struct.	Beh.	Non-context	Context	Total
Prepare candidate services	20	13703	703	13547	172	3453	31609
	40	24172	1328	25845	344	4437	56219
	60	33812	2078	37843	516	3906	78313
		R1	**R2**	**R3**	**R4**	**R5**	**Avg**
Unavailable (i)		109	110	78	93	157	109.4
New Service (ii)		1687	1782	1672	2015	1984	1828
Service change (iii)		1797	1781	1672	1656	1656	1712.6

Furthermore, someone should consider the longer term cost of the two modes of query execution. More specifically, assuming that the service associated with query Q4 becomes unavailable X times, in the pull mode of query execution the total cost of service discovery required to identify replacement services using Q4 will be X*36572 milliseconds (for 20 services), X*67383 milliseconds (for 40 services) and X*99093 milliseconds (for 60 services). In contrast, in the push query execution mode, the respective total times will be X*109.4 (see the average time needed for case (ii) across the different runs of Q4). Similar gains arise in the cases where a new service becomes available (X*1828) or there is a change in a service description (X*1712). The magnitude of gains becomes even more substantial as in the push query execution mode the above activities are executed in parallel to the application.

Finally, the results in Table 4 show that the time to identify a service due to unavailability (case (i)) of a service is smaller than the time to identify a service due to changes in a service (case (iii)) or the time to evaluate a new service that becomes available (case (ii)). This is because cases (ii) and (iii) require the re-execution of the query in order to calculate its distance with the changed/new service, whilst in case (i) a replacement service is taken from *Set_S*.

Note that, Table 4 presents no results related to changes in the context of the application environment. In this case, a new query must be created and evaluated against all the services in the registry. Therefore, the time to identify a service to replace an existing service in this case is equivalent to the time to execute a query in pull mode.

5.3 Discussion

Overall, the results of our experiments demonstrate that RSDF has promising performance since (a) the query execution time in both pull and push modes of execution increases linearly with the size of the service registry, and (b) the use of push query execution mode results in considerable performance gains, making this mode a pragmatic and realistic approach for runtime service discovery. Moreover, the experiments have demonstrated that the performance gains with the push mode of query execution and the proactive approach for identifying candidate replacement services in parallel to the execution of a service-based application provide a good support for continuously changes in service-based applications with respect to the various circumstances described in the paper.

The results of the experiments reported in Sect. 5.2 above are similar to the results of an earlier, albeit smaller scale, experiment that was reported in [62] and in which we had investigated the time required for executing structural, behavioural, contextual and non-contextual SerDiQueL queries in pull mode. More specifically, both experiments demonstrated the same relative costs of the executions of different parts of a query (i.e., the behavioural conditions are the most expensive to execute followed by structural and contextual/non contextual conditions). Both experiments have also demonstrated that the time taken to retrieve services from the registry was significantly larger than the time taken to execute the different types of matchings and that the execution time of each type of the structural, behavioural, non contextual and contextual matching increases linearly with the size of the registry. In the experiments we conducted in this paper, we have also investigated the time for executing queries in push mode and have demonstrated the gains that arise from this mode of execution. Gains from the execution of queries in push mode were also reported in [32], although a direct comparison with the results of this papers cannot be made as in [32], different querying scenarios and service data sets were used due to the need to incorporate service monitoring in the experiment.

Overall, further experimentation is needed to confirm our initial findings and investigate the performance of RSDF in different scenarios of context and service updates (e.g., high/medium/low frequency updates of service descriptions and context conditions).

It should also be noted that although proactive service discovery is essential for achieving efficient service replacement at runtime, it might also result in inefficient utilization of resources. This would happen in cases where proactively discovered services get replaced in the buffer set without being used. Over a time period T, the efficiency of resource utilization with proactive discovery can be measured by the formula:

$$U = \frac{T \times SRRR \times t_{match}}{T \times SRUR \times t_{match} + t_{init\text{-}RS}}$$

In this formula, SRRR is the service request replacement rate; SRUR is the service registry update rate; t_{match} is the average time required to match a query with a service;

and $t_{init\text{-}RS}$ is the time needed to build the initial copy of RS ($t_{init\text{-}RS}=R_{init} \times t_{match}$, where R_{init} is the number of services in the service registry at the time of the initial build of RS).

Efficient resource utilization arises when SRRR ≥ SRUR + R_{init}/T or when SRRR ≥ SRUR as T increases and, therefore, the factor R_{init}/T tends to zero. This means that the service replacement request rate must be higher or at least equal to the service registry update rate. Establishing whether the service request replacement rate is greater than or equal to the service registry update rate would require a long-term study. However, it is not unrealistic to expect that SRRR ≥ SRUR holds in the long term.

6 Related Work

Several approaches have been developed to support service discovery. Semantic matchmaking approaches constitute a significant category of them, based on explicit representation of semantics and logic reasoning [1][7][28][30]. The METEOR-S [1] system, for example, adopts a constraint driven service discovery where service requests are integrated into the composition process of a service-based application and [28] uses logic based approximate matching and IR techniques. The work in [7] considers composition fragments to support service composition and reuse. It uses description logic to identify fragments and semantic matchmaking of fragments and goals to support the discovery process. Note, however, that these approaches do not consider dynamic service discovery, neither support push mode query execution.

Other approaches for service discovery consider graph transformation rules [24], or behavioural matching [20][35][47]. The work in [24] is limited since it cannot account for changes in the order or names of the parameters. In [47], the authors use service behaviour signatures to improve service discovery. The works in [21] and [48] describe functional and quality characteristics of components and services as aspects and discovery is based on a formal analysis and validation of these descriptions. In [47] a query language based on first-order logic that focuses on properties of behavior signatures is used to support the discovery process. The work in [35] advocates the use of behavioral specifications represented in BPEL for service discovery in order to resolve ambiguities between requests and services, and uses a tree-alignment algorithm to identify matchings between requests and services. However, none of the above approaches supports proactive service discovery as ours. In [15] the authors propose a requirement-centric approach to support modeling, discovery, and selection of web-services. In this approach, the discovery process is based on keyword matching. The selection process is based on formal concept analysis in which services with common QoS properties are grouped together and organized into concept lattices.

In [56] the authors propose a monitorable contract model to support dynamic monitoring of business process and proactive detection of contract violations. The proactive detection is based on the use of guards of monitoring constraints that consider actions that have occurred, actions that have not occurred but are expected to occur, and actions that should not occur in the future. Other approaches have been proposed to support adaptation and changes in service-based applications in a reactive [2][3][18][26][31] or proactive way [11][25][29][34][53].

The reactive approaches propose changes in service composition based on pre-defined policies [3], self-healing of compositions based on detection of exceptions and repair using handlers [18], context-based adaptation of compositions using negotiation and repair actions [2], and key performance indicator (KPI) analysis and use of adaptation strategies based on KPI fulfillment [26]. The work in [18], advocates a model-based to support repair of faulty activities in service-based processes. The approach uses repair actions and plans that are generated by considering constraints of the process structure and dependencies among data.

The proactive approaches use semi-Markov models for prediction of performance failures and support self-healing of service compositions [11], event monitoring and machine learning techniques for prediction and prevention of SLA violations [29], testing techniques to anticipate problems in service-based applications and trigger adaptation requests [34][53], cross-layered adaptation strategies for software and infrastructure layers [22].

Several approaches have also been proposed to support context awareness in service discovery [6][10][42][58]. In [10], context information is represented by key-value pairs attached to the edges of a graph representing service classifications. This approach does not integrate context information with behavioural and quality matching and, context information is stored explicitly in a service repository that must be updated following context changes. In [6] queries, services, and context information are expressed in ontologies. The approach in [4] focuses on user context information (e.g. location and time) and uses it to discover the most appropriate network operator before making phone calls. The work in [58] locates components based on context-aware browsing. The above context-aware approaches support simple conditions regarding context information in service discovery, do not fully integrate context with behavioural criteria in service discovery, and have limited applicability since they depend on the use of specific ontologies for the expression of context conditions.

Another group of approaches have been proposed to support service selection based on trust and reputation of services [12][33][51][54][57]. In [12] users are responsible for providing ratings and expectation values on QoS attributes. The approach described in [57] uses a reputation manager to calculate reputation scores and assumes that service consumers will provide QoS requirements, weights to be associated to the reputation score, QoS scores, and ratings to assess the services. In [51] the authors describe an approach to service selection based on the user's perception of the QoS attributes rather than the actual attribute values. The work in [33] does not provide

ways of checking whether the same feedback in different websites is used more than once. The QoS-based service selection and ranking solution in [54] supports prediction of future quality of web services. The authors introduce a mechanism to avoid unfair ratings based on statistical analysis of the reports from users.

Query languages, other than *SerDiQueL*, have also been proposed to support services discovery [4][40][41][59]. These include BP-QL[4], a visual query language for BPEL. *SerDiQueL* also supports querying BPEL specifications. However, our work differs from BP-QL since it supports the specification of structural, quality, and contextual conditions in the query, and the behavioural conditions can be matched against other types of behavioural service specifications. The query language proposed in [41] is used to support composition of services based on user's goals. NaLIX[59], a language for querying XML databases based on natural language, has also been applied to service discovery. Keyword-based retrieval underpins some service registries available on the Internet (e.g. *seekda* [45] and *servicefinder* [46]). These approaches enable also discovery through service categories and the use of tags. These search modalities are easy to use and useful in design time service discovery but cannot offer the matching precision that is required in runtime service discovery that is executed to support automatic service replacement in applications. USQL (Unified Service Query language) [40] is an XML-based language enabling discovery based on syntactic, semantic, and quality of service search criteria that has some similarity to *SerDiQueL*. *SerDiQueL*, however, is more complete since it supports the specification of behavioral criteria for the services to be discovered, as well as context characteristics of services and application environments.

In summary, most of the proposed approaches support service discovery based on limited sets of service criteria and in reactive (pull) mode of query execution. Unlike them, RSDF supports proactive dynamic service discovery based on a flexible and comprehensive set of service and application criteria including not only structural and quality constraints but also functional and contextual characteristics. It also supports pull and push service discovery, resulting in more efficient service replacement during the execution of an application. Our approach uses fine grain quantification of similarities between queries and services based on distance measures.

In reference to our previous work on dynamic service discovery, the framework presented in this paper extends the work in [49] by introducing a new language for the specification of behavioural conditions in service discovery queries, and the work in [62] by introducing a new way of computing behavioural distances for the behavioural part of queries. As discussed earlier, the new query language allows the declarative specification of conditions regarding the behaviour of services as opposed to the procedural BPEL specifications used in earlier versions. Also the new algorithm for the computation of the behavioural distance verifies if the path representing behavioural conditions in the query can be matched to a path in the state machine representing the behavioural specification of a service by considering the semantic of the conditions (i.e., elements) used in the behavioural sub-query. In [62], the behavioural distance was computed by comparing state machines representing BPEL specifications of queries and services; verifying if the transition paths of the query state machine could be transformed into transition paths in the service state machine; and computing a penalty for transformations that were not exact. We should note that the new version of the framework that has been presented in this paper supports the execution of queries expressed in earlier versions except from their behavioural parts of queries.

Given that in registries there are often services that have no behavioural descriptions, in [32] we investigated the possibility of using a monitor component to verify the satisfiability of behavioural and contextual properties of services, expressed in SerDiQueL, against messages exchanged between service-based applications and their deployed services. A proof-of-concept implementation of this approach was presented in [32] but the experimental evaluation indicated deterioration in the performance of query execution when monitoring is used to verify behavioural subqueries.

7 Conclusions and Future Work

In this paper we have presented a proactive framework for dynamic service discovery, in which candidate services to replace existing services in a service-based application are identified in parallel to the execution of these applications. Our framework supports service discovery in both pull and push modes of query execution due to (a) unavailability or malfunctioning of services, (b) changes in the structure, functionality, quality, or context of the services, (c) changes in the context of the application environment, or (d) availability of new services.

The pull mode of query execution is performed by searching service registries to identify services to be bound to an application. The push mode of query execution is based on subscribed services and queries, as well as up-to-date sets of candidate services. In both pull and push query execution modes, a service is matched against a query based on computation of distances between query and service specifications. The framework uses complex queries expressed in an XML-based query language named *SerDiQueL*. The language allows the representation of structural, behavioural, quality, and contextual characteristics of services and applications.

A prototype tool has been implemented to illustrate and evaluate the framework. The evaluation has focused on the execution time of the retrieval process and has shown promising results (linear increase of discovery time with respect to service registry size and significant gains from the use of push mode query execution).

Planned future work on RSDF will be aimed at integrating it with active service registries [52], which can

push information about new services and service updates to clients, in order to eliminate the need for polling that is currently performed by the RSDF service listeners. We are also planning to extend service registry intermediaries to support other types of service registries and evaluate the efficiency of RSDF under different scenarios of changes in context and service information at runtime, as discussed at the end of Sect. 5.

Current work focuses on investigating the use of other forms of adaptation of service-based applications as, for example, replacing malfunctioning or unavailable services by compositions of services and/or changing the structure of the service workflow in a service-based application. This work focuses on expressing and checking security conditions about individual services during the discovery process, as part of SerDiQueL. Other extensions of the current work investigates the use of the service discovery approach to support adaptation of sevice-based applications based on QoS prediction techniques, analysis of dependencies between service operations, and the possibility of compensanting QoS and behavipoural violations by other operations in the composition yet to be executed. We are also investigating the development of tools on the top of RSDF that would allow system developers to specify and check the correctness of SerDiQueL queries.

ACKNOWLEDGMENT

This work has been partially funded by the European Commission initially as part of the F6 project GREDIA (F6-34363) and subsequently as part of the F7 project ASSERT4SOA (F7-257351).

REFERENCES

[1] R. Aggarwal, K. Verma, J. Miller, and W. Milnor. Constraint Driven Web Service Composition in METEOR-S, 2nd Int. Conf. on Services Computing, 2004.

[2] D. Ardagna, M. Comussi, E. Musi, B. Pernici, P. Plebani. PAWS: A Framework for Executing Adaptive Web-Service Processes. IEEE Software, 24 (6), 2007.

[3] L. Baresi, C, Ghezzi, S. Guinea. Towards Self-Healing Compositions of Services. Studies in Comp. Intel., v. 42, Springer 2007.

[4] L. Baresi, E. Di Nitto, C. Ghezzi, and S. Guinea. A Framework for the Deployment of Adaptable Web Service Compositions. Service Oriented Computing and Applications J., 1(1), 2007.

[5] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying Business Processes. 32nd Int. Conf. on Very Large Data Bases, 2006.

[6] F. Bormann, et al, Towards Context-Aware Service Discovery: A Case Study for a new Advice of Charge Service, 14th IST Mobile and Wireless Communications Summit, June 2005.

[7] C. Bouhini, F. Lecue, N. Mehandjiev, O. Boissier. Discovery and Selection of Web Services Fragments for Re-Composition. IEEE Int. Conf. on Service-Oriented Computing and Applications (SOCA 2010), 2010.

[8] BPEL4WS.http://www128.ibm.com/developerworks/library/specification/ws-bpel/

[9] CoDAMoS. www.cs.kuleuven.ac.be/cwis/research/distrinet/projects/CoDAMoS/ontology/

[10] S. Cuddy, M. Katchabaw, and H. Lutfiyya. Context-Aware Service Selection Based on Dynamic and Static Service Attributes. IEEE Int. Conf. on Wireless and Mobile Computing, Networking and Communications, 2005.

[11] Y. Dai, L. Yang, B. Zhang. QoS-Driven Self-Healing Web Service Composition Based on Performance Prediction. Journal of Computer Science and Technology, 24(2), March 2009.

[12] V. Deora, J. Shao, W.A.Gray, N. J. Fiddian. A Quality of Service Management Framework Based on User Expectations. 1st Int. Conf. on Service Orienting Computing, 2003.

[13] C. Doulkeridis, N. Loutas, and M. Vazirgiannis. A System Architecture for Context-Aware Service Discovery. Electronic Notes of Theoretical Computer Science 146(1): 101-116 (2006).

[14] J. Dolley, A. Zisman, G. Spanoudakis. Runtime Service Discovery for Grid Applications. In Grid Technology for Maximizing Collaborative Decision Management and Support: Advancing Effective Virtual Organizations, 2009.

[15] M. Driss, N. Moha, Y. Jamoussi, J.M. Jezequel, H.H.B. Ghezala. A Requirement-Centric Approach to Web Service Modeling, Discovery and Selection. 8th Int. Conf. on Service Oriented Computing, 2010.

[16] DSD. Dynamic Service Discovery Framework Project, http://www.soi.city.ac.uk/~zisman/DSD_Project

[17] eXist. http://exist.sourceforge.net

[18] G. Friedrich, M. Fugini, E. Mussi, B. Pernici, and G. Tagni. Exception Handling for Repair in Service-Based Processes. IEEE Transactions on Software Engineering, 36(2), 2010.

[19] GREDIA project. http://www.gredia.eu.

[20] D. Grirori, J.C. Corrales, and M.Bouzeghoub. Behavioral Matching for Service Retrieval, Int. Conf. on Web Services, 2006.

[21] J. Grundy and G. Ding. Automatic Validation of Deployed J2EE Components Using Aspects. IEEE 16th Int. Conf. on Automated Software Engineering, 2001.

[22] S. Guinea, G. Kecskemeti, A. Marconi, and B. Wetzstein. Multi-layered Monitoring and Adaptation. 9th Int. Conf. on Service Oriented Computing, December 2011.

[23] I. Horrocks, P.F. Patel-Schneider, and F. van Harmelen. From SHIQ and RDF to OWL: The Making of A Web Ontology Language, J. of Web Semantics, 1(1), 7-26, 2003.

[24] U. Keller, R. Lara, H. Lausen, A. Polleres, and D. Fensel. Automatic Location of Services, Eur. Semantic Web Conf., 2005.

[25] N. Jun, Z. Bin, Z. Xiamgyu, Z. Zhiliang, L. Dancheng. Two-Stage Adaptation for Dependable Service-Oriented System. Int. Conf. on Service Sciences, 2010.

[26] R. Kazhamiakin, B. Wetztein, D. Karastoyanova, M. Pistore, and F. Leymann. Adaptation of Service-based Applications Based on Process Quality Factor Analysis. ServiceWave 2009.

[27] J. Kim, J. Lee, B. Lee. Runtime Service Discovery and Reconfiguration using OWL-S based Semantic Web Service. 7th IEEE Int. Conf. on Computer and Information Technology, 2007.

[28] M. Klusch, B. Fries, and K. Sycara. Automated Semantic Web Service Discovery with OWLS-MX, Int. Conf. on Autonomous Agents and Multiagent Systems, 2006.

[29] P. Leitner, A. Michlmayr, F. Rosenber, and S. Dustdar. Monitoring, Prediction and Prevention of SLA Violations in Composite Services. IEEE Int. Conf. on Web Services, 2010.

[30] L. Li and I. Horrock. A Software Framework for Matchmaking based on Semantic Web Technology, WWW Conference Work. on E-Services and the Semantic Web, 2003.

[31] K.J Lin, J. Zhang, Y. Zhai, and B. Xu. The Design and Implementation of Service Process Reconfiguration with End-to-end QoS Constraints in SOA. Journal of Service Oriented Computing and Applications, vol 4, 2010.

[32] K. Mahbub, G. Spanoudakis, and A. Zisman. A Monitoring Approach for Runtime Service Discovery, Automated Software Engineering Journal, 18(2): 117-161, 2011.

[33] L. Meng, Z. Junfeng, W. Lijie, C. Sibo, and X. Bing. CoWS: An Internet-Enriched and Quality-Aware Web Services Search Engine. IEEE Int. Conf. on Web Services, 2011.

[34] A. Metzer, O. Sammodi, K. Pohl, M. Rzepka. Towards Proactive Adaptation with Confidence Augumenting Service Monitoring with Online Testing, Software Engineering for Adaptive and Self-managing Systems, South Africa, May 2010.

[35] R. Mikhaiel and E. Stroulia. Interface- and Usage-aware Service Discovery, 4th Int. Conf. on Service Oriented Computing, 2006.

[36] O Moser, F. Rosenberg, S. Dustdar. Non-Intrusive Monitoring and Service Adaptation for WS-BPEL. 17th Int. World Wide Web Conference, 2008.

[37] J. Morato, M.A. Marzal, J. Llorens, and J. Moreiro. WordNet Application, 2nd Global Wordnet Conference, 2004.

[38] H. Niu and Y. Park. An Execution-based Retrieval of Object-Oriented Components. 37th ACM Southeast Reg. Conf., 1999.

[39] OCL. http://www.omg.org/docs/ptc/03-10-14.pdf

[40] M. Pantazoglou, A. Tsalgatidou, and G. Athanasopoulos. Discovering Web Services in JXTA Peer-to-Peer Services in a Unified Manner. 4th Int. Conf. on Service Oriented Computing, 2006

[41] M. Papazoglou, M. Aiello, M. Pistore, J. Yang. XSRL: A Request Language for web services, citeseer.ist.psu.edu/575968.html

[42] P. Pawar and A. Tokmakoff. Ontology-based Context-aware service discovery for pervasive environments, IEEE Int. Work. On Service Integration in Pervasive Environment, June, 2006

[43] PayPal. http://www.paypal.com

[44] SECSE Project. http://secse.eng.it

[45] Seekda, http://webservices.seekda.com/

[46] Servicefinder, http://demo.service-finder.eu/search

[47] Z. Shen and J. Su. Web Service Discovery based on Behavior Signatures. 3rd Int. Conf. on Service Computing, 2005.

[48] S. Singh, J. Grundy, J. Hosking, J. Sun. An Architecture for Developing Aspect-Oriented Web Services, 3rd Eur. Conf. in Web Services, 2005.

[49] G. Spanoudakis, K. Mahbub, and A. Zisman. A Platform for Context-Aware Run-time Service Discovery, 2007 IEEE Int. Conf. on Web Services, 2007

[50] G. Spanoudakis and A. Zisman. Discovering Services during Service-based System Design using UML, IEEE Transactions of Software Engineering, 36(3): 371-389, 2010

[51] A. Srivastava and P. G. Sorenson. Service Selection based on customer Rating of Quality of Service Attributes. IEEE Int. Conf. on Web Services, 2010.

[52] M. Treiber and S. Dustdar, Active web service registries, IEEE Internet Computing, 11(5): 66–71, 2007

[53] D. Tosi and G. Denaro and M. Pezzè. Towards Autonomic Service-Oriented Applications. Int. Journal of Autonomic Computing (IJAC), 2009, pp. 58—80

[54] L.Vu,M.Hauswirth,andK.Aberer. QoS based Service Selection and Ranking with Trust and Reputation Management. Proc. of the Cooperative Information System Conf., 2005.

[55] WSDL. http://www.w3.org/TR/wsdl

[56] L. Xu and M.A. Jeusfeld. A Concept for Monitoring of Electronic Contracts. 15th Conf. on Advanced Information Systems Engineering (CAiSE'03), Austria, June 2003.

[57] Z. Xu, P. Martin, W. Powley, and F. Zulkernine. Reputation-Enhanced QoS-based Web Services Discovery. IEEE Int. Conf. on Web Services, 2007.

[58] Y. Ye and G. Fischer. Context-Aware Browsing of Large Component Repositories. 16th Int. Conf. on Automated Software Engineering, 2001.

[59] L.Y. Yunyao, H. Yanh, and H. Jagadish,. NaLIX: an Interactive Natural Language Interface for Querying XML, SIGMOD 2005.

[60] A. Zisman, K. Mahbub, and G. Spanoudakis. A Service Discovery Framework Based on Linear Composition, IEEE 2007 Int. Service Computing Conference, 2007.

[61] A. Zisman, G. Spanoudakis, and J. Dooley. Proactive Runtime Service Discovery, 2008 Int. Service Computing Conf. 2008.

[62] A. Zisman, G. Spanoudakis, and J. Dooley. A Framework for Dynamic Service Discovery, Int. IEEE Conf. on Automated Software Engineering, 2008.

Andrea Zisman holds PhD, MSc, and BSc degrees in Computer Science. She is a Professor at City University London. Andrea has been research active in the areas of software and service engineering where she has published extensively. Web page: http://www.soi.city.ac.uk/~zisman

George Spanoudakis holds BSc, MSc and PhD degrees in Computer Science. He is a Professor at City University London. His research interests are in the area of service oriented computing and software engineering where he has published extensively. Web page: http://www.soi.city.ac.uk/~gespan.

James Dooley received his BSc in Robotics and Intelligent Machines from the University of Essex (UK). He worked for BT research and on European projects.

Igor Siveroni holds a PhD and MSc degrees in Computer Science and a BSc in Industrial Engineering. His research interests are in program analysis. He has participated in European and EPSRC research projects.